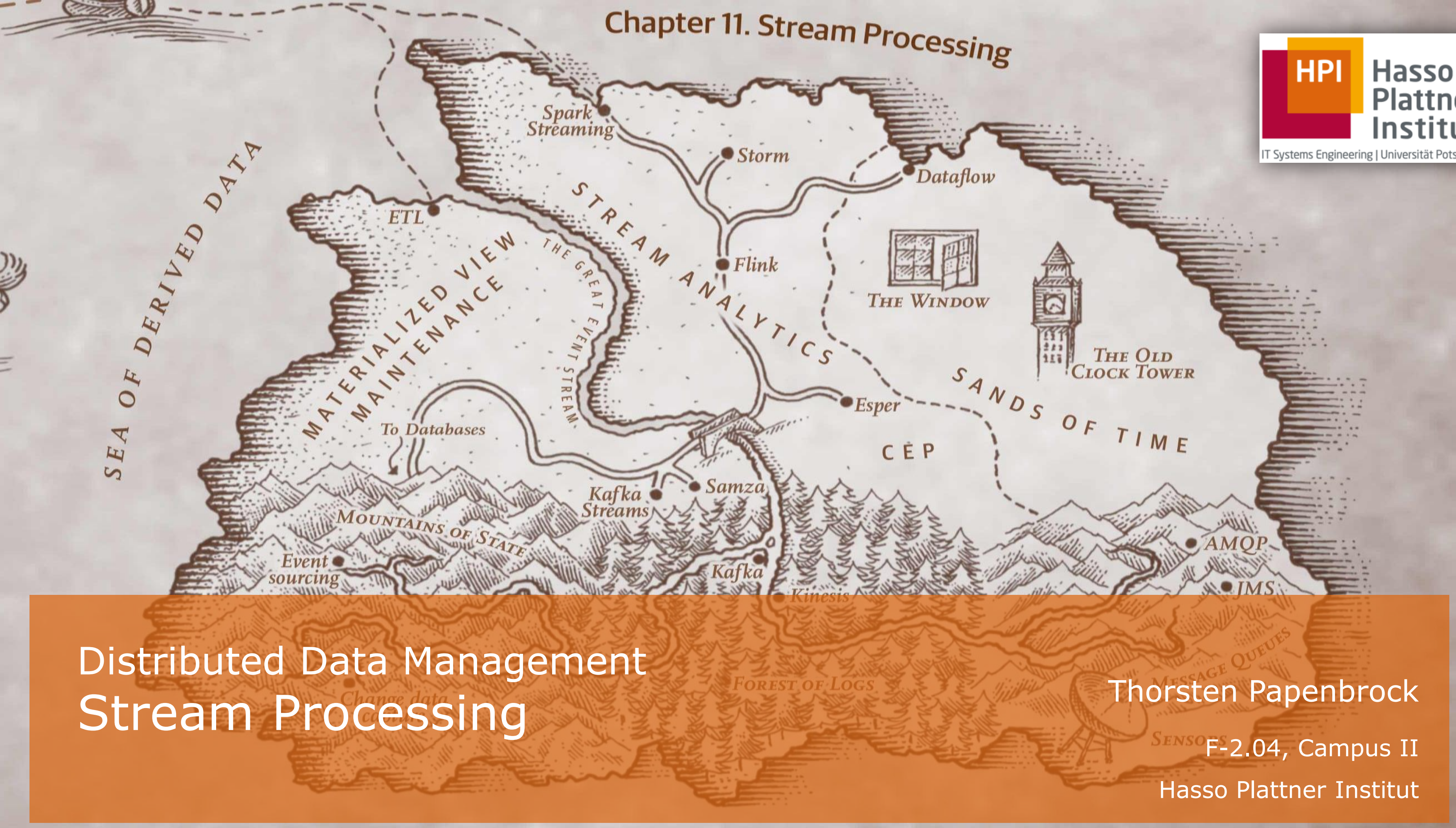


Distributed Data Management
Stream Processing

Thorsten Papenbrock
F-2.04, Campus II
Hasso Plattner Institut

Types of Systems

Services (online systems)

- Accept requests and send responses
- Performance measure: response time and availability
- Expected runtime: milliseconds to seconds

OLTP

Batch processing systems (offline systems)

- Take (large amounts of) data; run (complex) jobs; produce some output
- Performance measure: throughput (i.e., data per time)
- Expected runtime: minutes to days

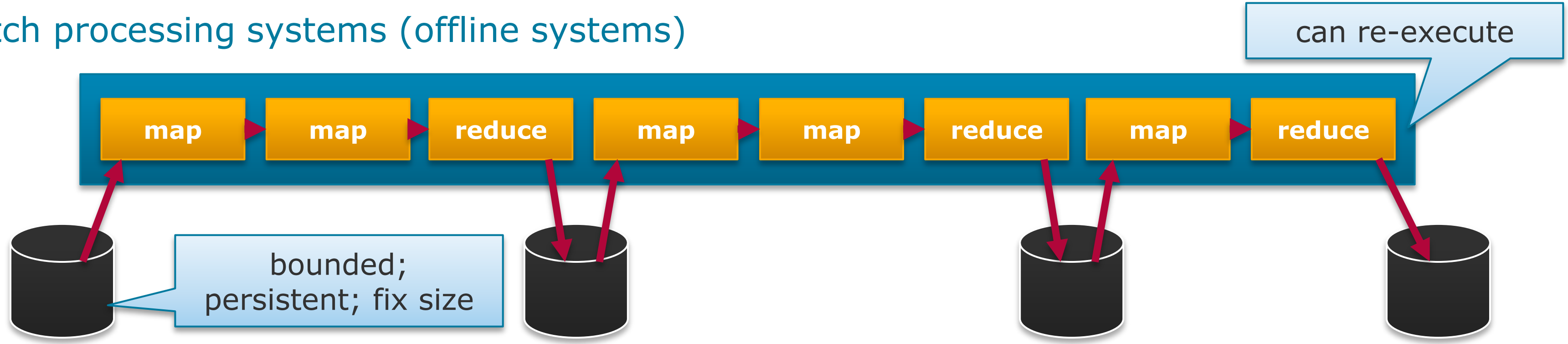
Stream processing systems (near-real-time systems)

- Consume volatile inputs; operate stream jobs; produce some output
- Performance measure: throughput and precision
- Expected runtime: near-real-time (i.e., as data arrives)

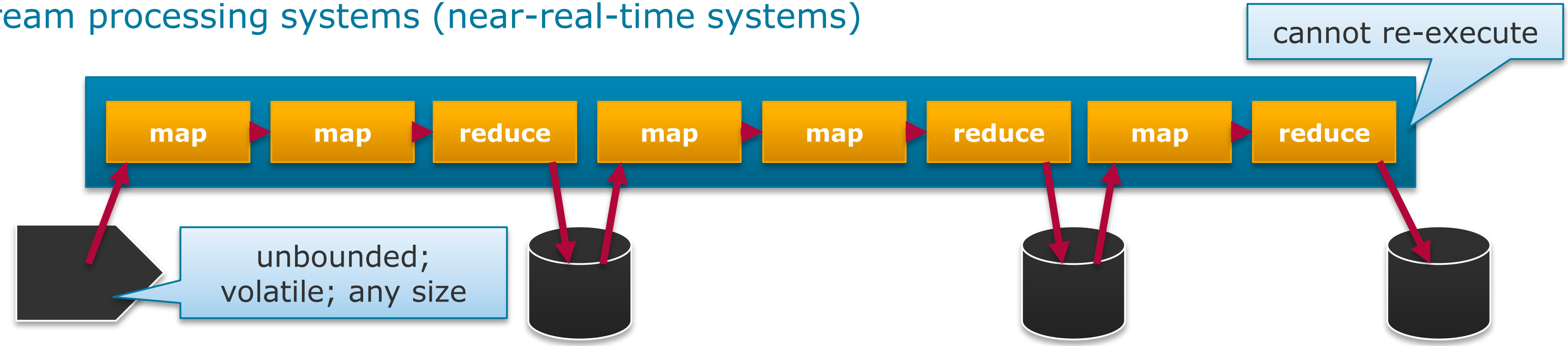
OLAP

Types of Systems

Batch processing systems (offline systems)

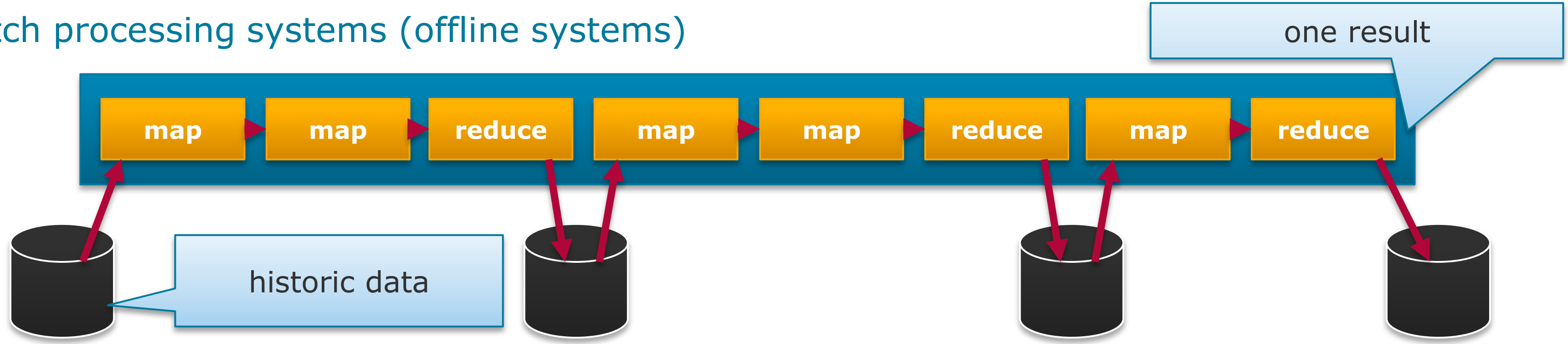


Stream processing systems (near-real-time systems)

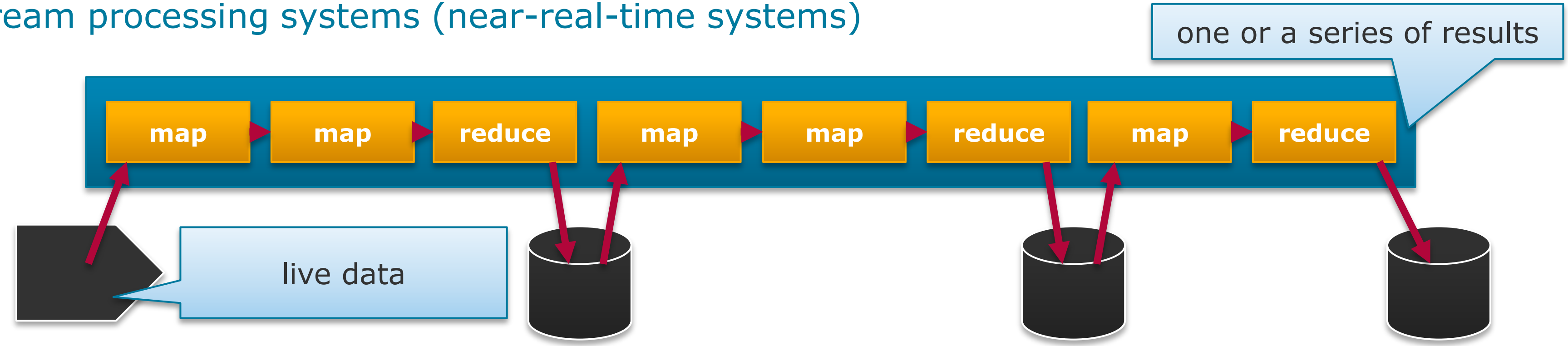


Types of Systems

Batch processing systems (offline systems)



Stream processing systems (near-real-time systems)



Use Cases for Streaming Data

Hasso Plattner Institut

Sensor Processing

- Continuous and endless readings by nature

Process Monitoring

- Side effects of processes that are continuously observed

Location Tracking

- Continuous location updates of certain devices

Log Analysis

- Digital footprints of applications that grow continuously

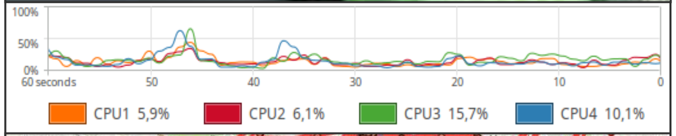
User Interaction

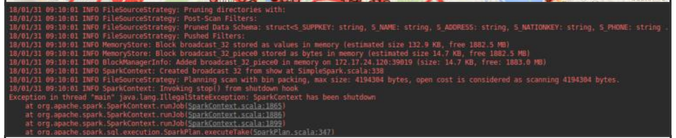
- Continuous and oftentimes bursty click- and call-events

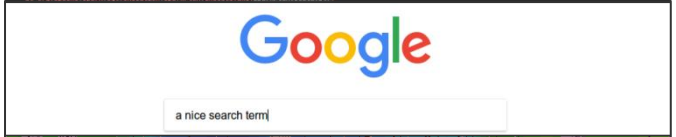
Market and Climate Prediction

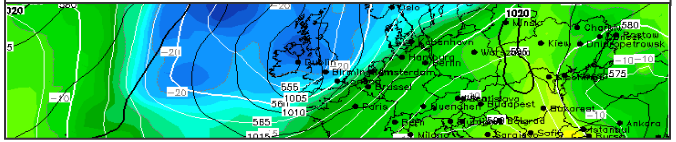
- Changing stock market prices and weather characteristics

…





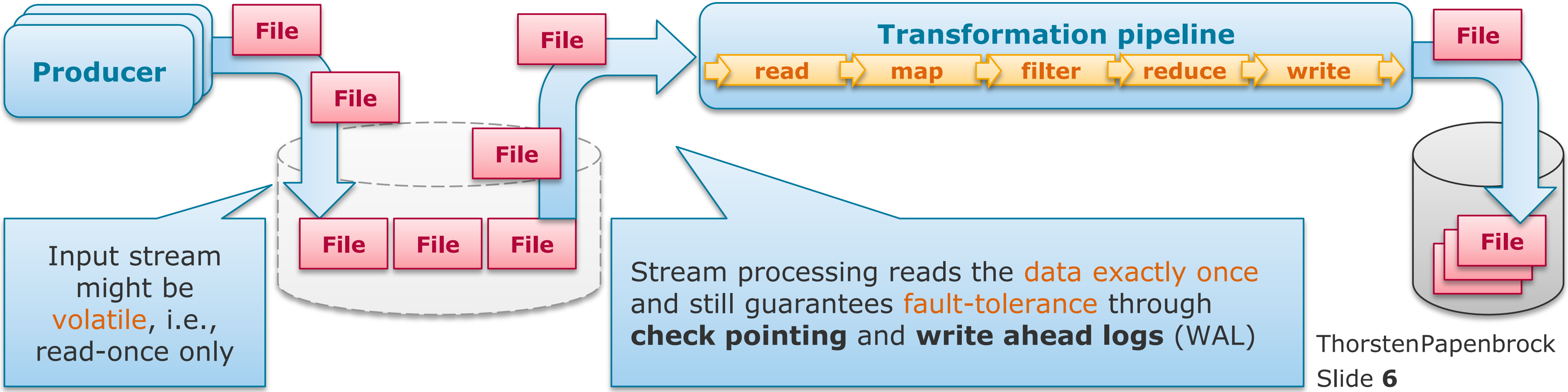




Spark Streaming (Recap)

Batched Stream Processing

- Reasons:
  - Incremental processing: start processing data that is still being written to
  - Latency reduction: pipeline data to maximizing resource utilization



Streams

Data Stream

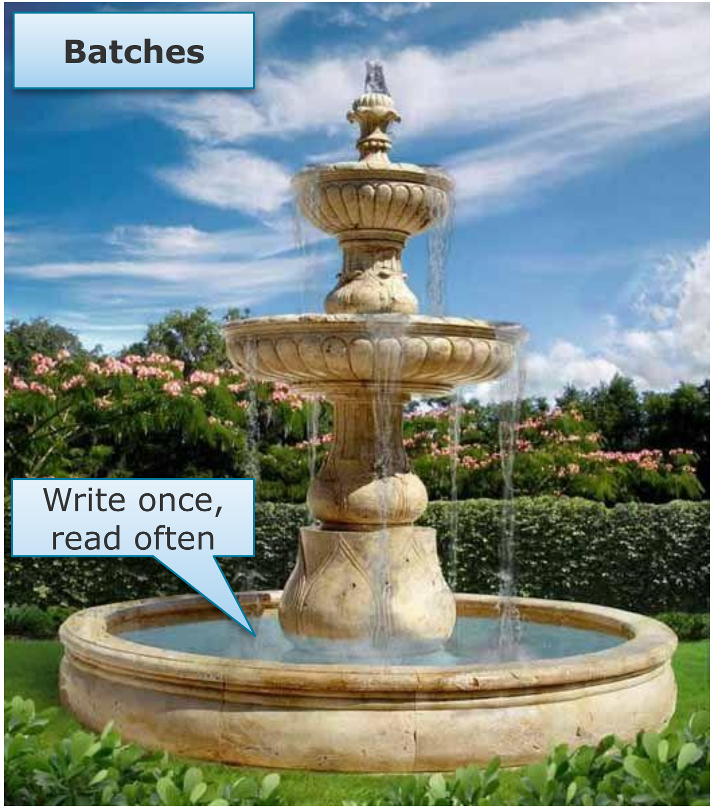
- Any data that is incrementally made available over time
- Examples:
  - Unix `stdin` and `stdout`
  - Filesystem APIs (e.g. Java's `FileInputStream`)
  - Online media delivery (audio/video streaming)
- Creation from …
  - static data: files or databases (read records line-wise)
  - dynamic data: sensor readings, service calls, transmitted data, logs, …

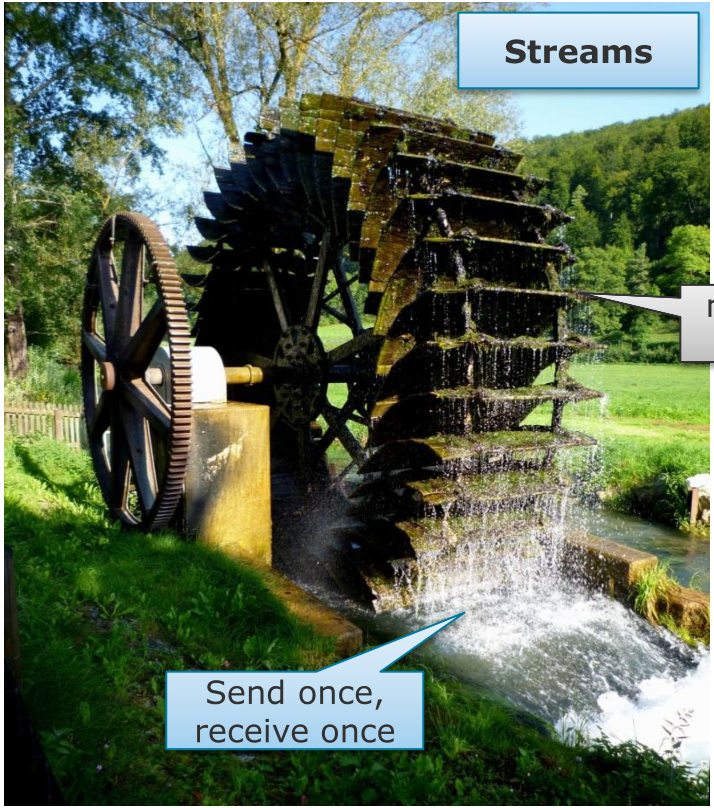
Event

- = an immutable record in a stream (often with timestamp)
- "Something that happened"
- Encoded in Json, XML, CSV, … maybe in binary format

Any format that allows incremental appends

Batch vs. Stream





maybe multiple
receivers

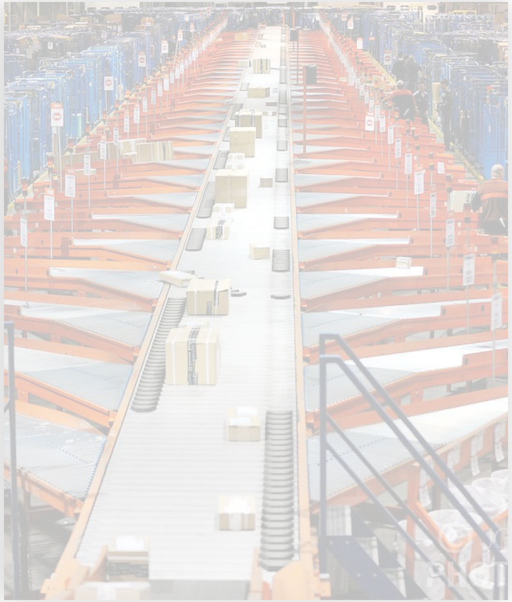
Overview

Stream Processing

Transmitting Event Streams

Databases and Streams

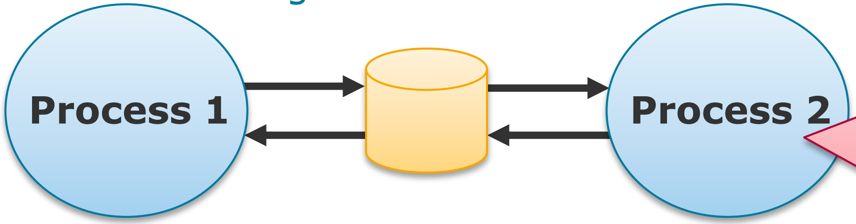
Processing Streams



Transmitting Event Streams

Event Transmission

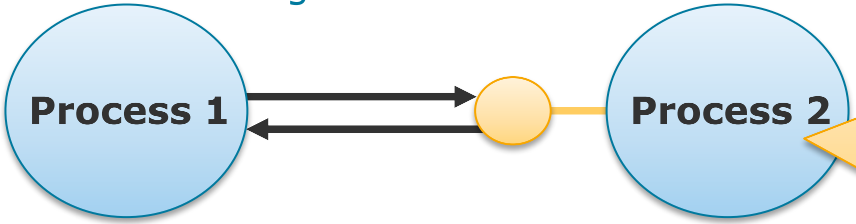
Dataflow Through Databases



Process 2 needs to poll the database for updates

- ➢ bad performance
- ➢ slow event propagation

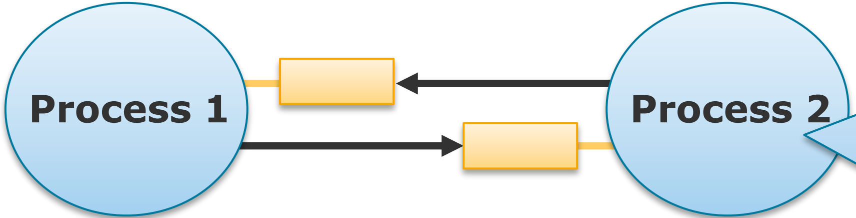
Dataflow Through Services



Working speed of process 2 determines stream speed

- ➢ maybe bad performance
- ➢ ok-isch event propagation

Message-Passing Dataflow



Asynchronous messaging and notification about new events

- ➢ good performance
- ➢ fast event propagation

Message-Passing Dataflow (Recap)

Communication

- Objects send messages to other objects via queues.

Message

- Container for data (= events)
- ➢ Often carries metadata (sender, receiver, timestamp, …)

Message queue

- Data structure (queue or list) assigned to communicating object(s)
- ➢ Enqueues messages in order of arrival
- ➢ Buffers incoming messages for being processed
- ➢ Notifies subscribers if new messages are available

Message Congestion

What if the stream producer is faster than the stream consumer(s)?

a) Drop messages
   - Delete messages that cannot be accepted.
   - ➢ Ok for use cases where timeliness is more important than completeness (e.g. for processing of sensor readings)

b) Buffer messages
   - Store messages in a cache until resources are available.
   - ➢ Ok to capture load spikes and if there is no constant overload that fills up buffers permanently (e.g. for user activity event streams)

c) Apply backpressure
   - Block the sender until resources are available.
   - Ok if the sender can be blocked and if the stream is not generated from outside (e.g. for reading a file as a stream from disk)

Most messaging systems use a mix of all three options.

Messaging Faults

What if nodes crash or temporarily go offline?

a) Fault ignorance

- Failed messages are lost.
- ➢ Ensures optimal throughput and latency

b) Fault tolerance

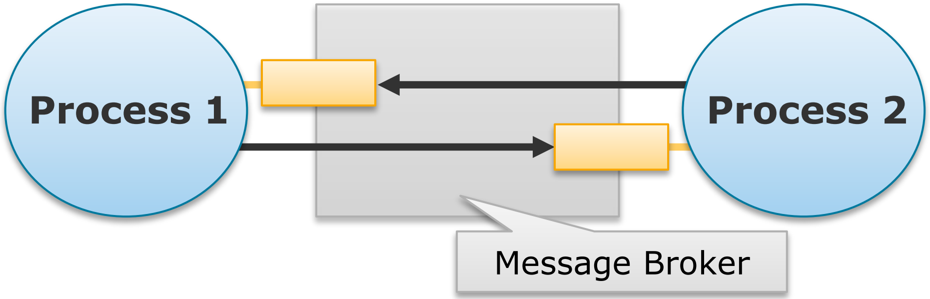
- Failed messages are recovered from checkpoints (disk or replicas).
- ➢ Ensures messaging reliability

More on fault tolerance later!

Message Brokers (Recap)

Message Broker

- Also called message queue or message-oriented middleware
- Part of the message-passing framework that delivers messages from their sender to the receiver(s)
- ➢ Maintains queues that sender can post messages to
- ➢ Notifies subscribers on new messages
- ➢ Resolves sender an receiver addresses
- ➢ Applies binary encoding when necessary
- ➢ Define the …
  - message congestion strategy
  - messaging fault strategy



If it blocks and persists,
then it is a database, right?

Message Brokers vs. Databases

Message Broker

- Short lived messages
  - Delete messages once successfully transmitted
- Small working set
  - If the number of pending messages increases, the performance drops (disk!)
- Subscription-based retrieval
  - Deliver messages to all subscribers of a queue
- Push client communication
  - Knows clients and initiates communications

Database

- Long-term persisted records
  - Store records until explicitly deleted
- Large working set
  - If the number of records increases, the performance is hardly affected
- Query-based retrieval
  - Read records upon client query using indexes
- Pull client communication
  - Clients are unknown and initiate communications

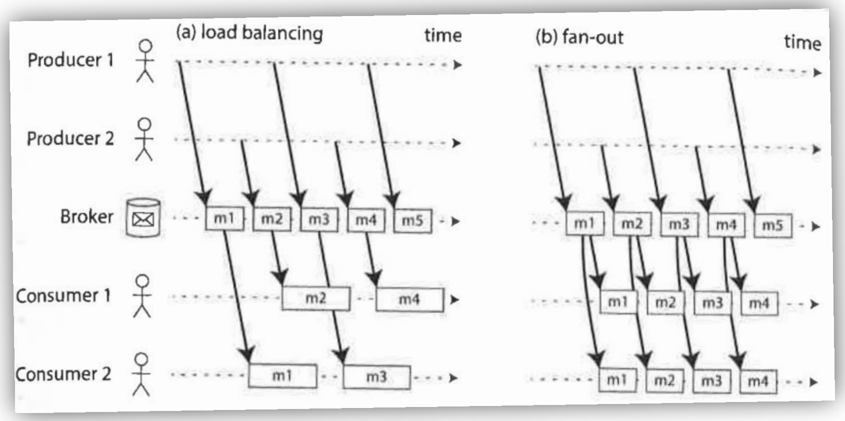
Message Brokers

Routing

- Producer send messages to queues.
- Message Broker notifies one or many consumers about such deliveries.
- Routing strategies:
  - a) One-to-one messages (Load Balancing)
    - Messages are routed to one subscriber
    - ➢ For data parallelism

    Partition input stream
  - b) One-to-many messages (Fan-out)
    - Messages are routed to all subscribers
    - ➢ For task parallelism

    Replicate input stream



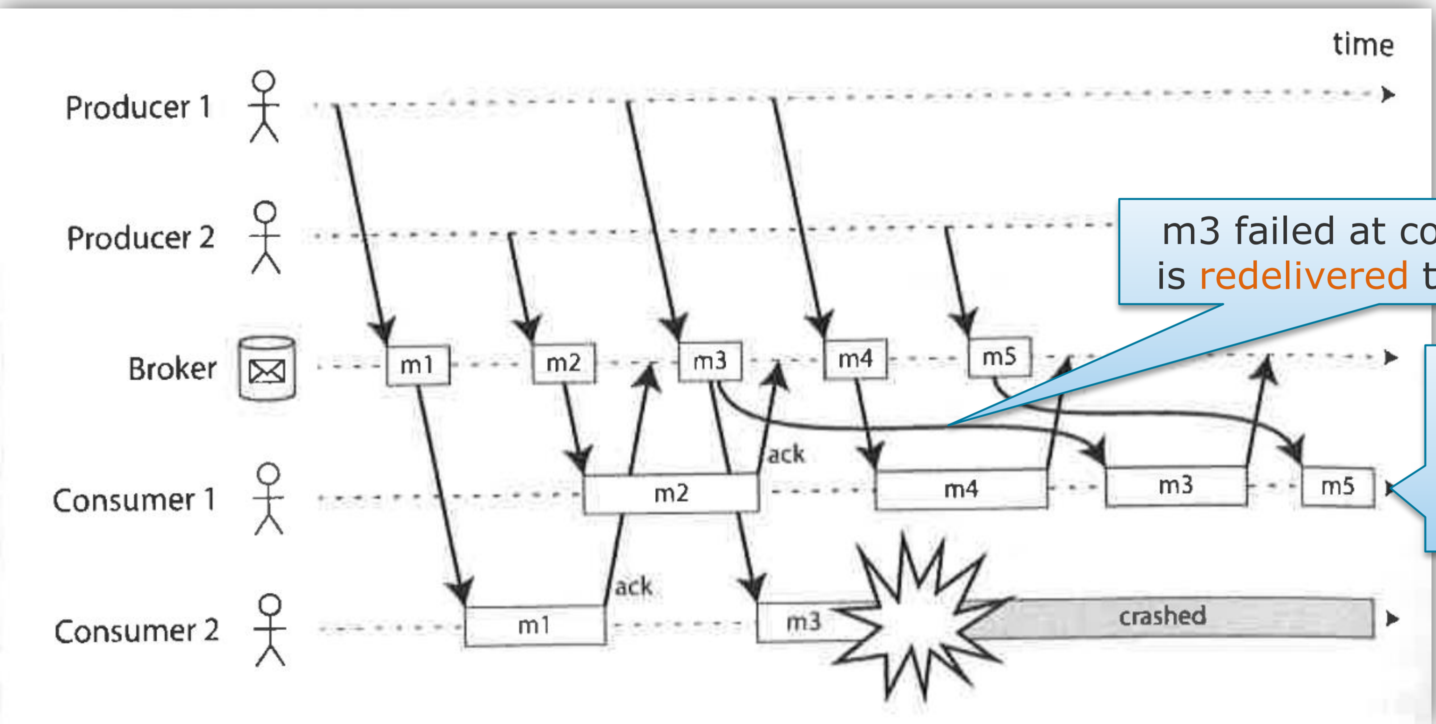
Message Brokers

Fault tolerance

- Acknowledgement:
  - Consumer send an acknowledgement to the Message Broker when they successfully received/completed a message.
  - Message Broker removes any completed message from its queues.
- Redelivery:
  - If acknowledgement fails to appear, the Message Broker redelivers it (perhaps to a different consumer).

Message Brokers

Fault tolerance



m3 failed at consumer 2 and is redelivered to consumer 3.

m3 is preserved but stream at consumer 3 is now out-of-order!

Transmitting Event Streams

Message Brokers: Persist or Forget

- Keep entire message stream (until reaching size or time limit)
- No need to track consumers
- Let consumers go back in time
  - ➢ Database-like
- Log-based Message Broker (e.g. Kafka, Kinesis or DistributedLog)

kafka

- Remove processed messages from stream (immediately after acknowledgement)
- Track consumers to forget old content
- The past is past
  - ➢ Volatile, light-weight
- Queue-based Message Brokers (e.g. RabbitMQ, ActiveMQ or HornetQ)

RabbitMQ

Log-based Message Broker

- Message broker that persist messages as logs on disk (distributed, replicated)
- Logs are immutable and append-only
  - Excellent sequential read performance
  - Support parallel, conflict-free reading by multiple clients
- Uncontrolled one-to-many messaging (we do not know who will read a message)
- Replicated Logs
  - For fault tolerance and better parallel read performance
  - Leader-based (to avoid complex replication protocols)
- Partitioned Logs
  - For parallel writes
  - Message ordering guaranteed only within a partition
    (not between partitions)
  - Partitioning strategies:
    - round-robin, load, partition size, semantic keys, …

Queue-based Message Broker

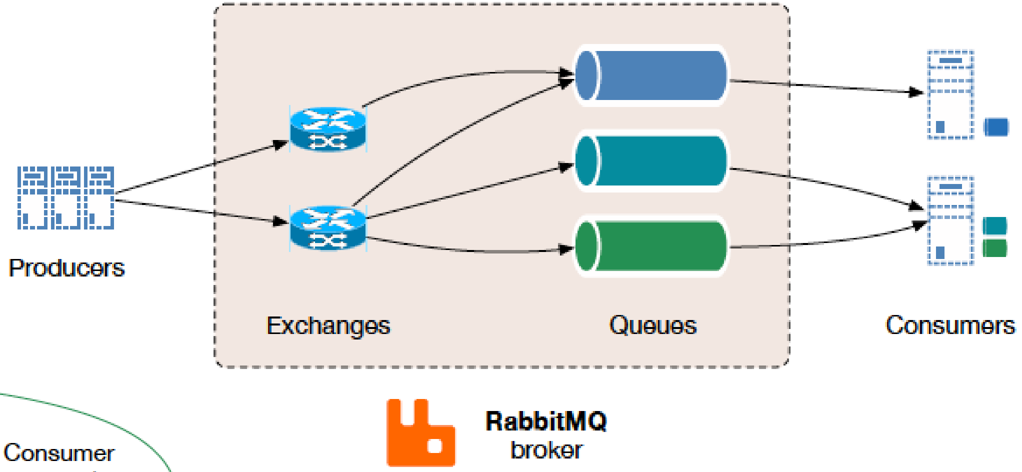
[1]Java Message Service (JMS) 2.0 Specification
[2]Advanced Message Queuing Protocol (AMQP) Specification

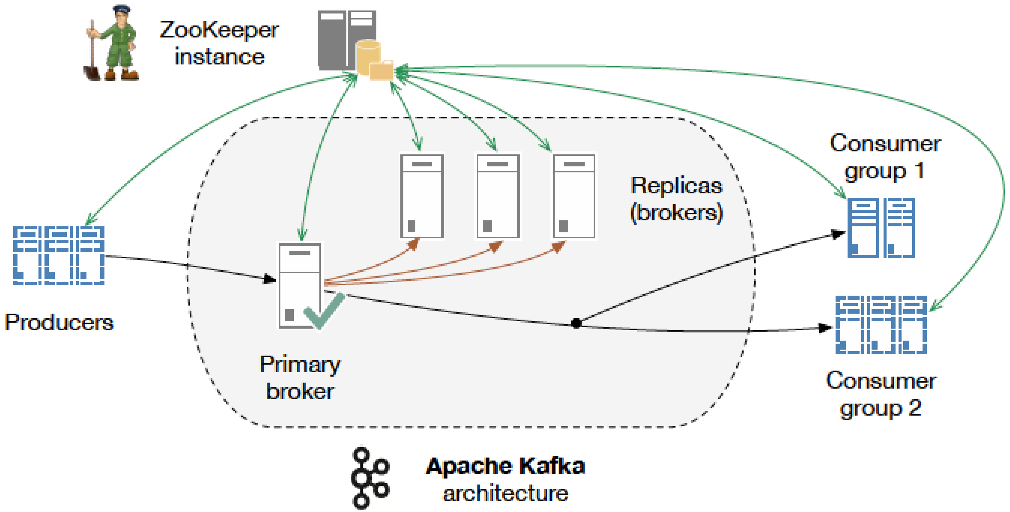
- Message broker that store messages in queues (distributed, replicated)
- Queues are mutable (usually in-memory) FIFO list data structures
  - Append messages at the end
  - Remove messages from the top
- Controlled one-to-one or one-to-many messaging (usually via JMS[1] or AMQP[2] protocols)
- Replicated/Mirrored Queues
  - For fault tolerance and availability only
    (no performance gain, because all replicas need to do all appends/removes)
  - Leader-based (to avoid complex replication protocols)
- No partitioning for queues
  - Create multiple queues manually if needed
- Reliability:
  - Send-and-acknowledge handshake with clients
    (keep messages until successfully acknowledged)

Message Brokers: Persist or Forget

https://content.pivotal.io/blog/understanding-when-to-use-rabbitmq-or-apache-kafka

http://kth.diva-portal.org/smash/get/diva2:813137/FULLTEXT01.pdf





Message Brokers: Persist or Forget

Persist | Forget

Keep entire message stream
(until reaching size or time limit)

No need to track consumers

Let consumers go back in time

- Database-like

Log-based Message Broker
(e.g. Kafka, Kinesis or DistributedLog)

kafka

Remove processed messages from stream
(immediately after acknowledgement)

Track consumers to forget old content

The past is past

- Volatile, light-weight

Queue-based Message Brokers
(e.g. RabbitMQ, ActiveMQ or HornetQ)

RabbitMQ

Kafka

Hasso Plattner Institut

Topics and Partitions

- Topics are logical groupings for event streams.
    - e.g. click-events, temperature-readings, location-signals
    - Every topic is created with a fixed number of partitions.
- Partitions are ordered lists of logically dependent events in a topic.
    - e.g. click-events by user, temperature-readings by sensor, location-signals by car
    - Provide “happens-before semantic” for these events
    - Order is valid within each partition, not across different partitions.
    - Are accessed sequentially
        - Producers write new events sequentially.
        - Consumers read events sequentially.
    - Purpose:
        - Parallelism: to read a topic in parallel
        - Load-balancing: to store the events of one topic on multiple nodes

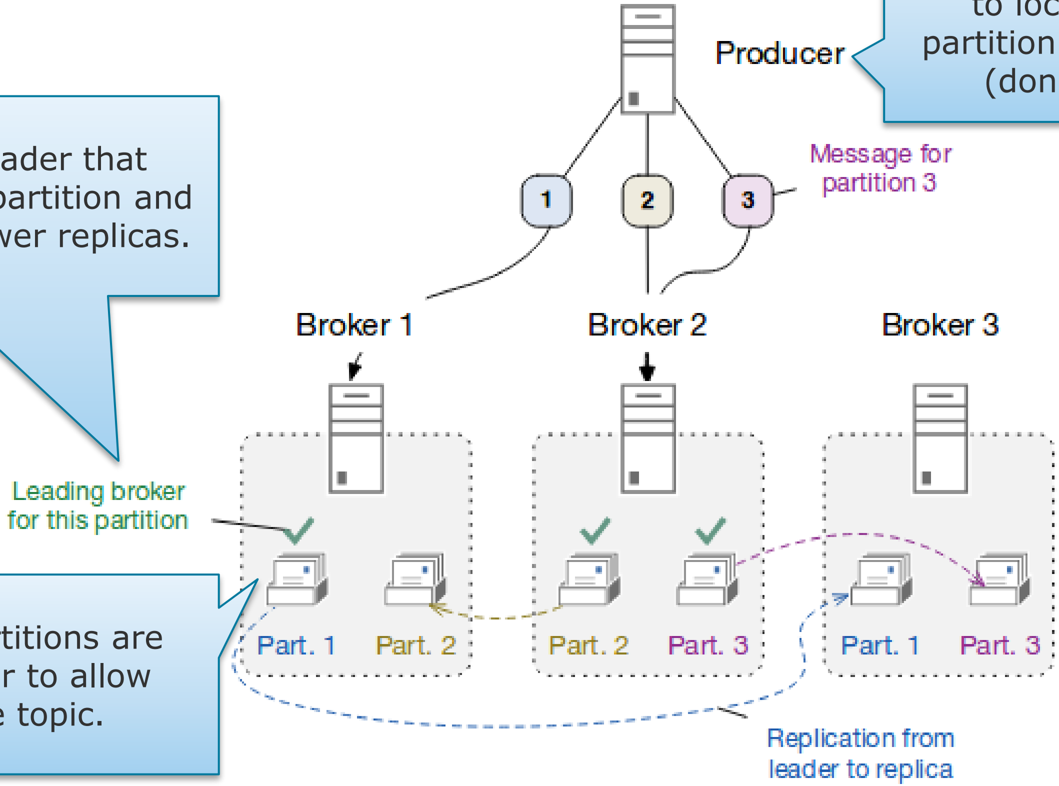
In many cases, event ordering is not a concern and partitions are simply arbitrary splits of a topic (for parallelization and load-balancing)

Kafka

Topics and Partitions

Every partition has a leader that accepts all writes to that partition and forwards them to its follower replicas.

A producer can ask any broker to locate the leader of a partition that it wants to write (done via ZooKeeper).



Leaders for different partitions are distributed in the cluster to allow parallel writes to one topic.

Kafka

Producers and Consumers

- Producers
  - Post to concrete partitions within a topic (only one leader can take these posts).
  - Define a Partitioner-strategy (on the producer side) to decide which partition is next.
    - Round-Robin Partitioner-strategy is used by default.
    - Custom Partitioner-strategies let producers define semantic grouping functions.
- Consumers
  - Read concrete partitions within a topic (all broker with that partition can take these reads).
  - Hold an offset pointer for every partition that they read (on consumer side).
  - Poll and wait (no callback registration)

"Kafka does not track acknowledgments from consumers […]. Instead, it *allows* consumers to use Kafka to track their position (offset) in each partition."

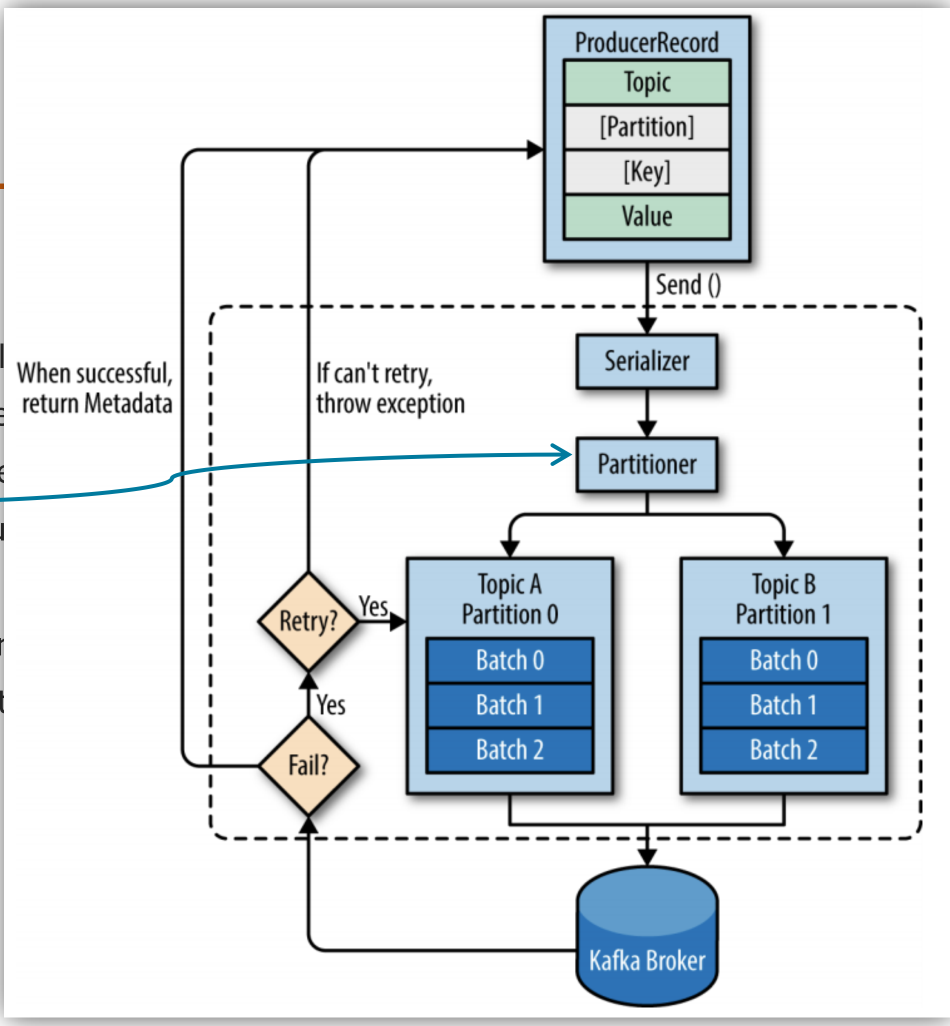
(Book: Kafka - The Definite Guide)

Transmitting Event Streams

Kafka

Producers and Consumers

- Producers
  - Post to concrete partitions within a topic (onl
  - Define a Partitioner-strategy (on the produce
    - Round-Robin Partitioner-strategy is use
    - Custom Partitioner-strategies let produ
- Consumers
  - Read concrete partitions within a topic (all br
  - Hold an offset pointer for every partition that
  - Poll and wait (no callback registration)

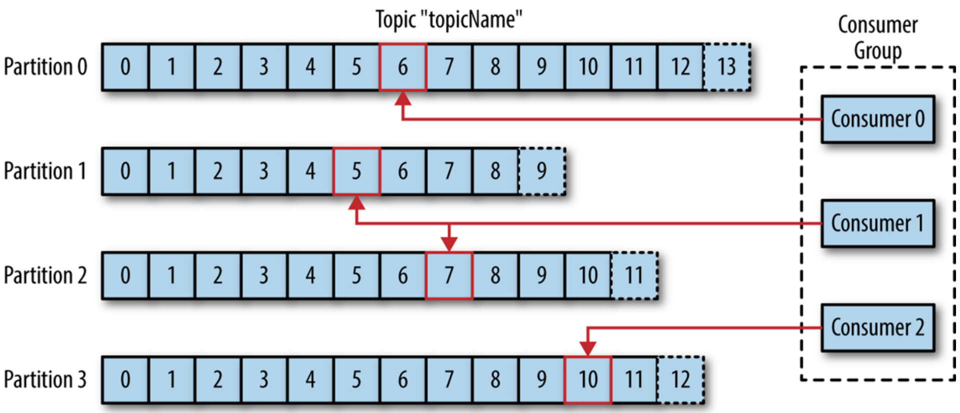


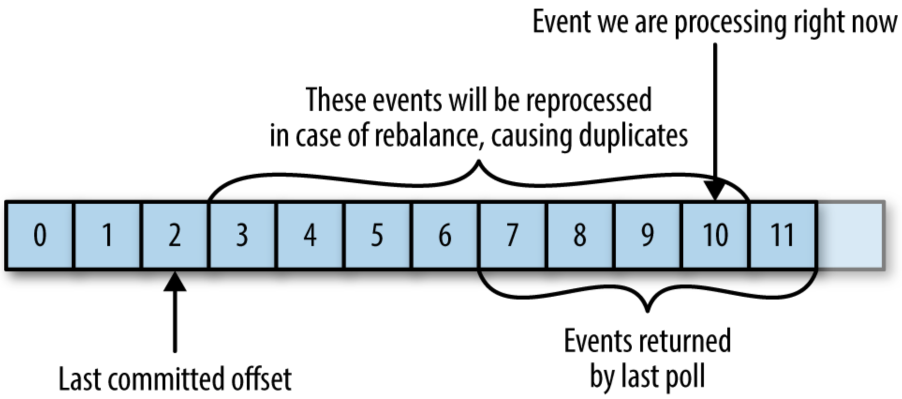
Kafka

Producers and Consumers

And in this way, Kafka kind of knows its consumers …

- Consumer Groups
  - A group of consumers that processes all events of one topic in parallel.
  - The offsets for a consumer group can be managed by Kafka on server side.
    - A dedicated group coordinator manages offsets, membership, scheduling etc.
    - Consumer commit successfully processed offsets to the group coordinator so that the coordinator can re-assign partitions to consumers.





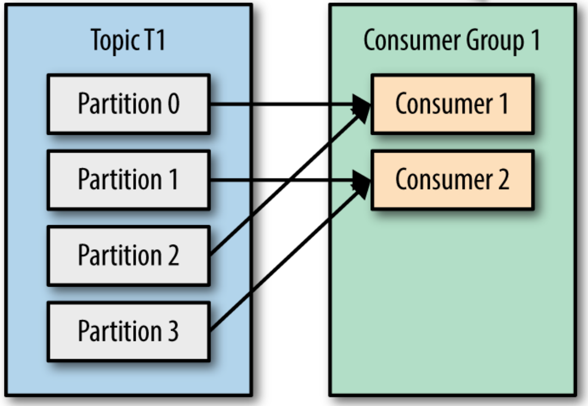
Transmitting Event Streams

Kafka

Producers and Consumers

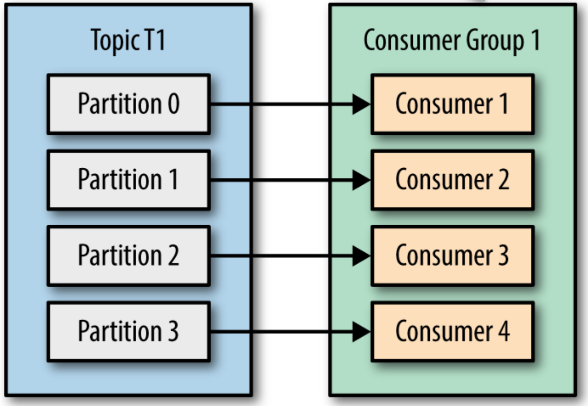
#partitions > #consumer

- Consumer take multiple partitions and process them alternatingly.



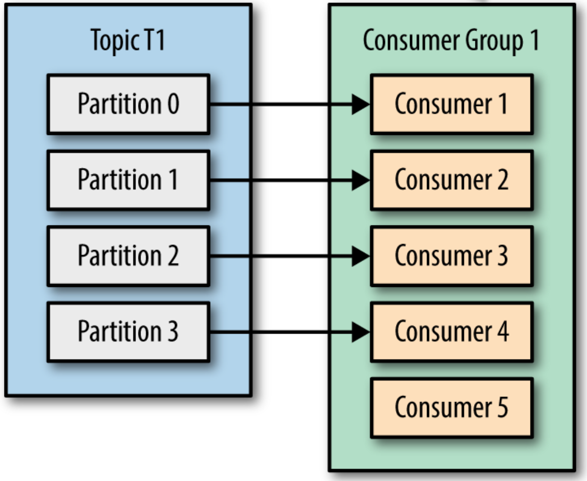
#partitions = #consumer

- Every consumer takes one partition; maximum parallelism.



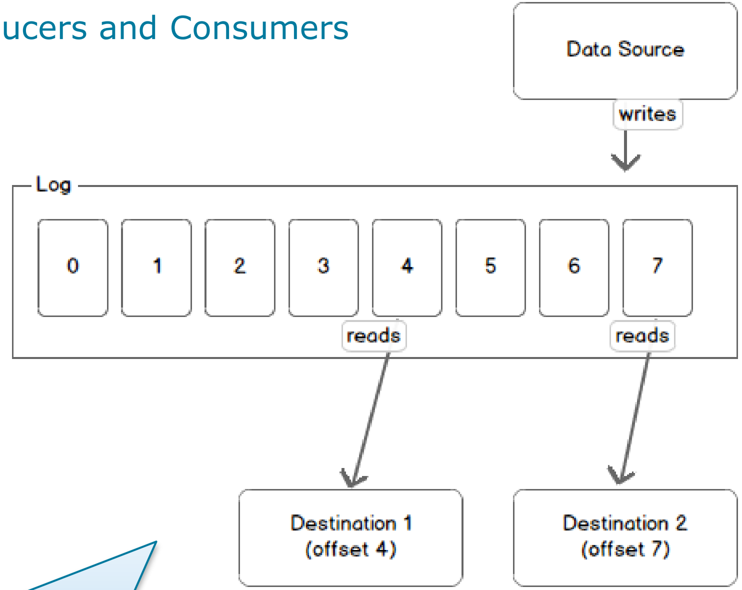
#partitions < #consumer

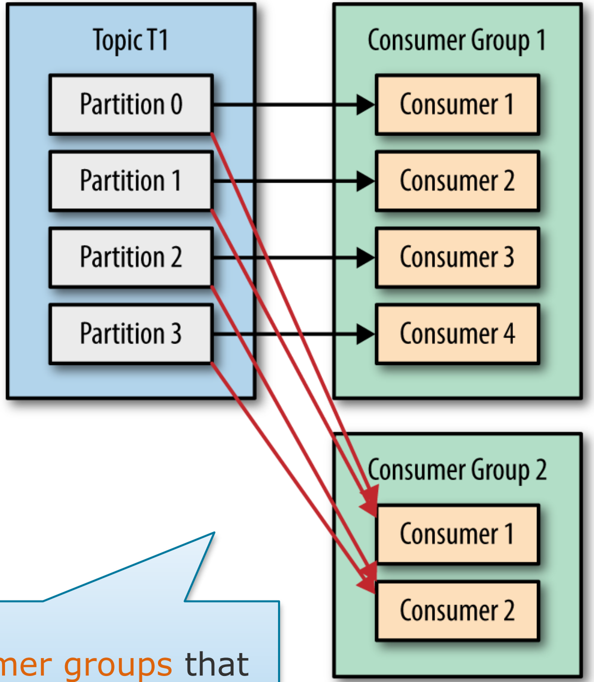
- Some consumers idle, because the group reads every partition exactly once.



Kafka

Producers and Consumers



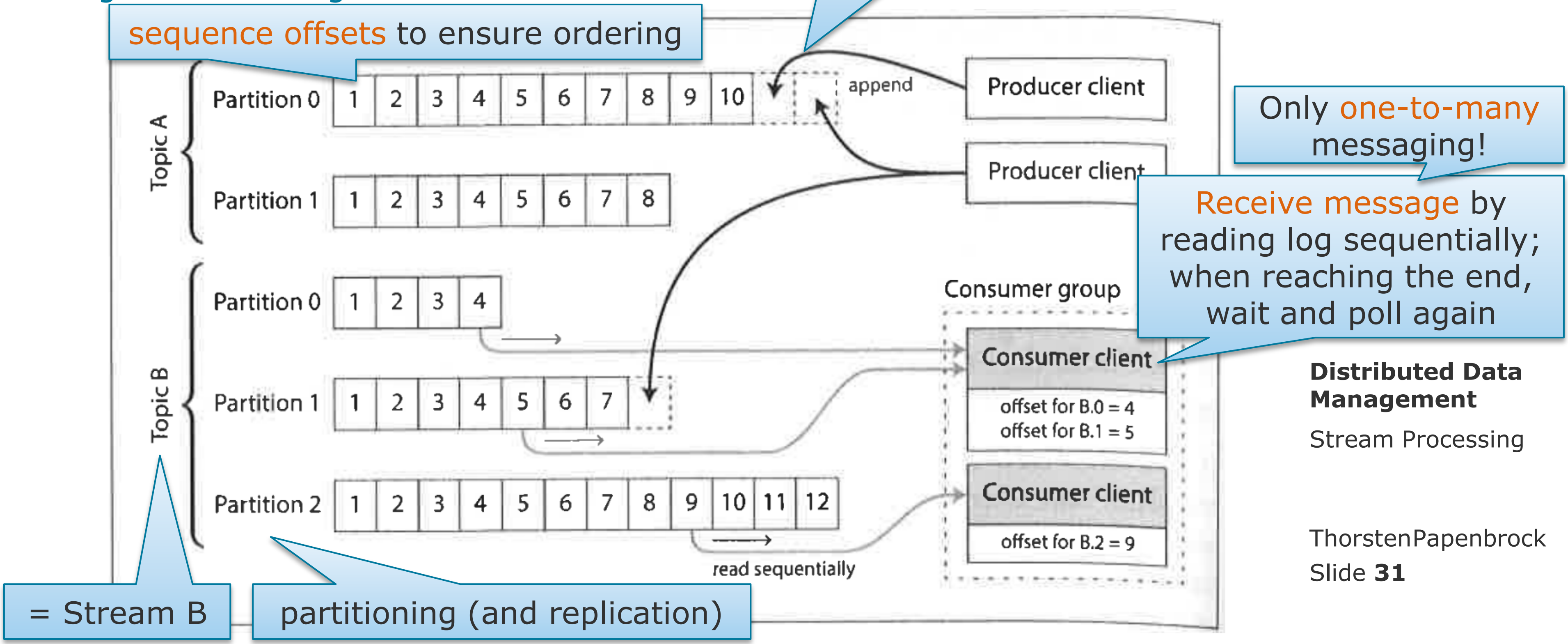


Different consumers that read the same partition in parallel and at different locations.

Different consumer groups that read same partitions in parallel (and at different locations).

Kafka

Log-based Message Broker



Transmitting Event Streams

Kafka

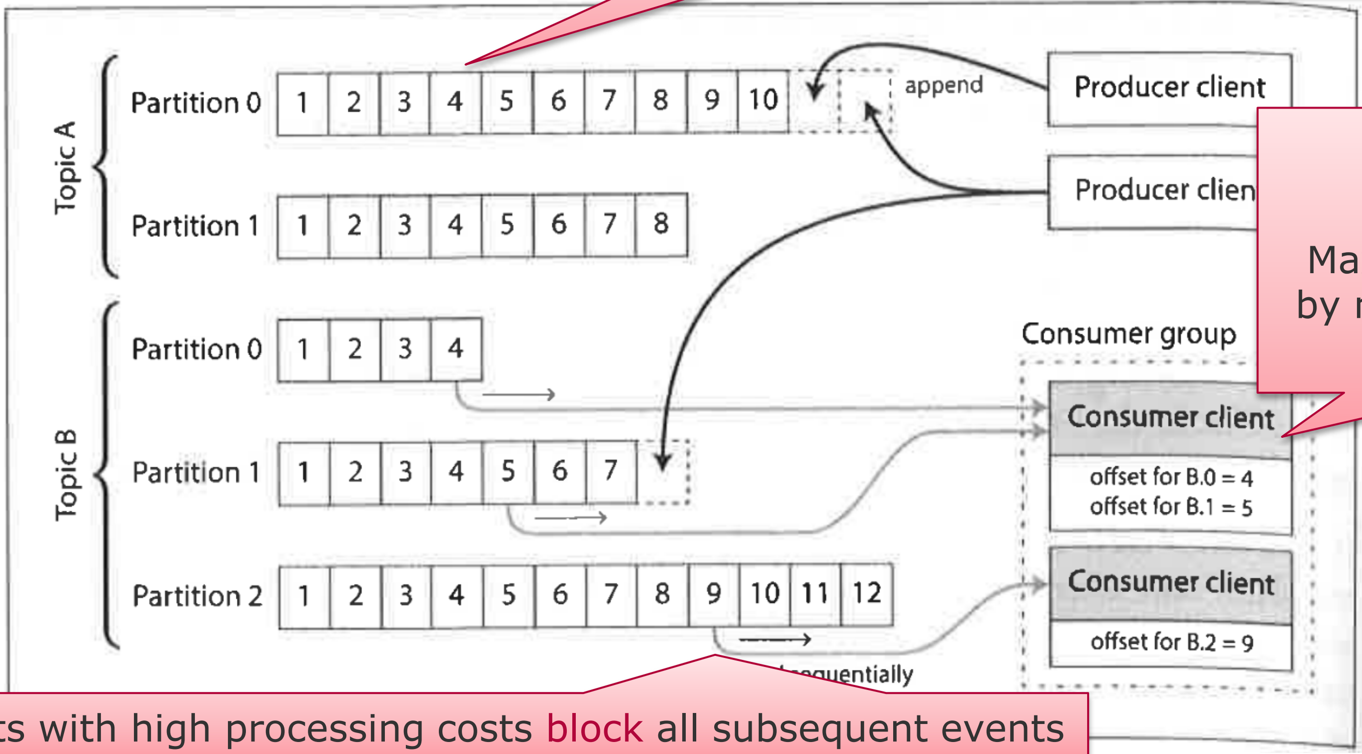
Log-based Message Broker

Example:

6 TB of disk capacity (= log size)
150 MB/s write throughput

11 h until an event is forgotten
(at maximum event throughput!)

Storing a history for events **costs memory**



No one-to-one scheduling:

Max **parallelism bound** by number of partitions in a topic!

Events with high processing costs **block** all subsequent events

Kafka

Kafka APIs

- Communication with Kafka happens via a specific APIs.
- The API can manage the specifics of the reading/writing process transparently.
  - e.g. offset-tracking (consumers) and partition-scheduling (producers)
- Two options:
  - A rich API that offers high abstraction, but limited control functions.
  - A low-level API that provides access to offsets and allows consumers to rewind them as the need.

Event lifetime

- Configurable:
  - By time of event
  - Max partition size

Kafka

HPI Hasso Plattner Institut

Optimizations that make Kafka fast:

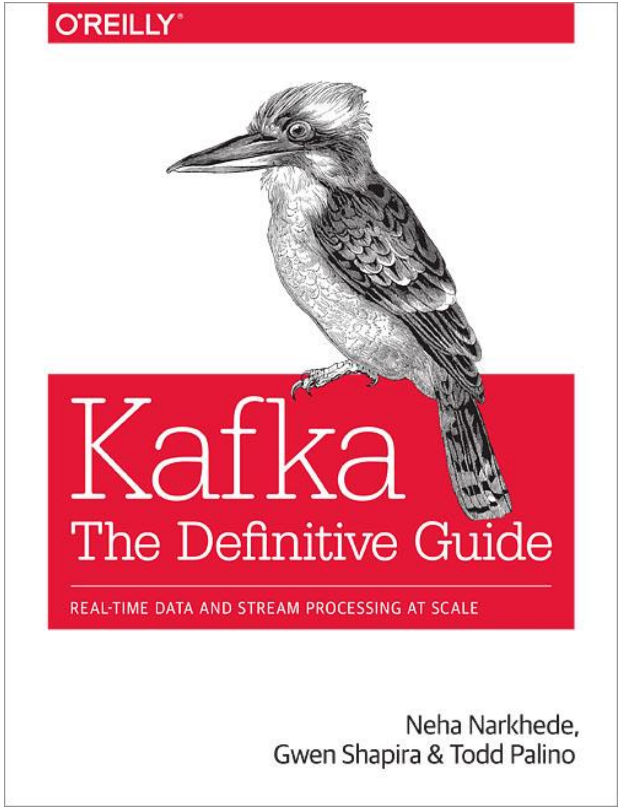
- Sequential I/O:
  - Sequential writes avoid disk seek times.
  - Exclusive write access to logs avoids blocking (one writer per log).
  - Sequential reads enable pre-fetching and caching of messages.
- Minimal serialization/deserialization:
  - Standardized binary formats let producers, brokers and consumers use the same data representations without individual modification.
- Zero-copy policy:
  - Data exchange completely in kernel space without copying it to user space avoids costly kernel-space to/from user-space copy processes
    (due to standardized formats, there is no need to copy messages into user space).
- Batch processing:
  - Batching of data reduces network calls and improves sequential writes.
  - Compression of batches (with LZ4, SNAPPY or GZIP) leads to better compression ratios.

Kafka

Further reading

- Kafka: The Definitive Guide

- https://www.oreilly.com/library/view/kafka-the-definitive/9781491936153/



Transmitting Event Streams

Message Brokers: Persist or Forget

- Keep entire message stream (until reaching size or time limit)
- No need to track consumers
- Let consumers go back in time
  - ➢ Database-like
- Log-based Message Broker (e.g. Kafka, Kinesis or DistributedLog)

- Remove processed messages from stream (immediately after acknowledgement)
- Track consumers to forget old content
- The past is past
  - ➢ Volatile, light-weight
- Queue-based Message Brokers (e.g. RabbitMQ, ActiveMQ or HornetQ)

Use if **throughput** matters, event processing costs are similar and the **order of messages** is important

Use if **one-to-one scheduling** is needed, **event processing costs differ** and the order of messages is insignificant

Message Brokers: Persist or Forget

Persist ⟷ Forget

- Keep entire message stream
  (until reaching size or time limit)
- No need to track consumers
- Let consumers go back in time
  - ➢ Database-like
- Log-based Message Broker
  (e.g. Kafka, Kinesis or DistributedLog)

Use if **throughput** matters,
event processing costs are similar and
the **order of messages** is important

Wait **throughput**?

Yes, because …

- ➢ dumping events to storage instead of routing them to consumers is faster.
- ➢ broker does not need to track acknowledgements for every event
  (only consumers track their queue offset).
- ➢ broker can utilize batching and pipelining internally.

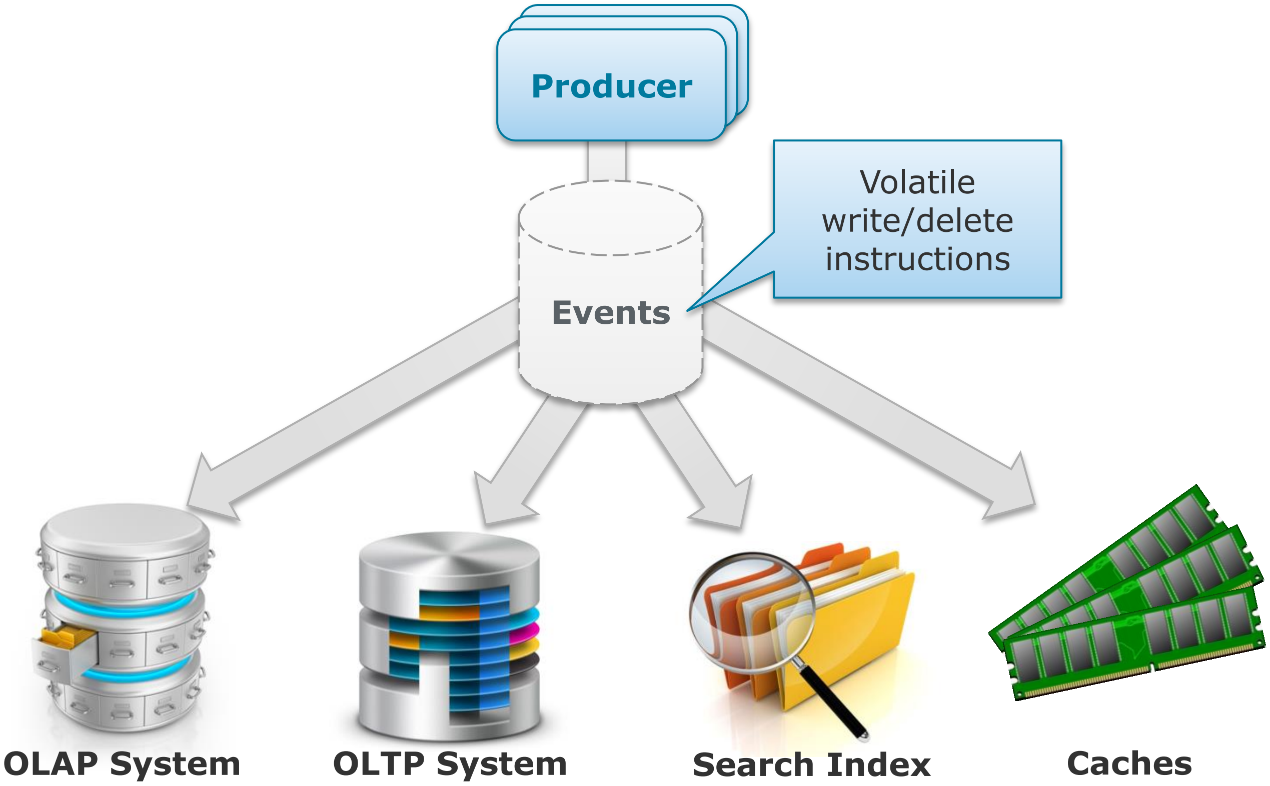
Stream Processing

Transmitting Event Streams

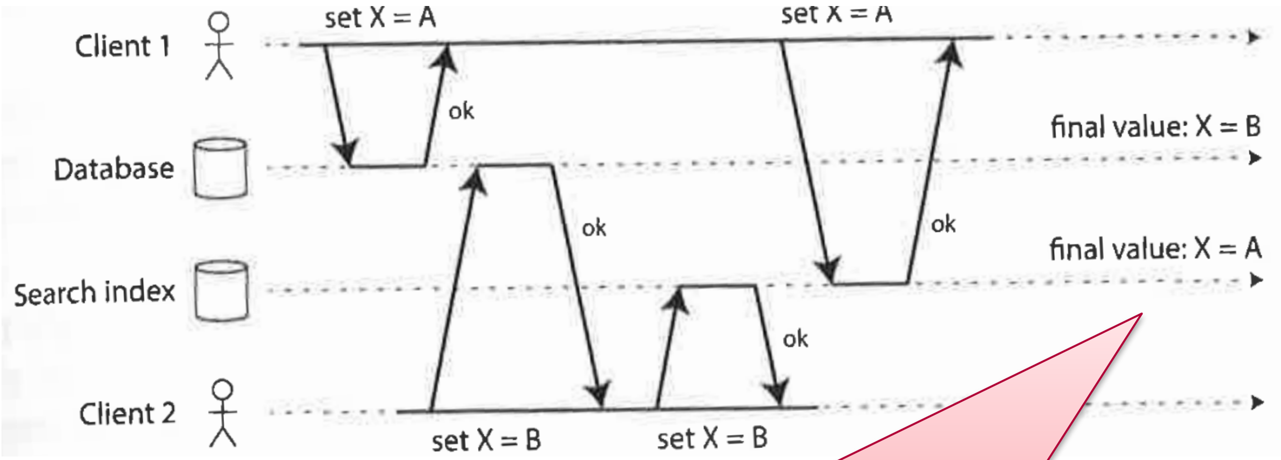
Databases and Streams

Processing Streams

Data Storage – Keeping Systems in Sync



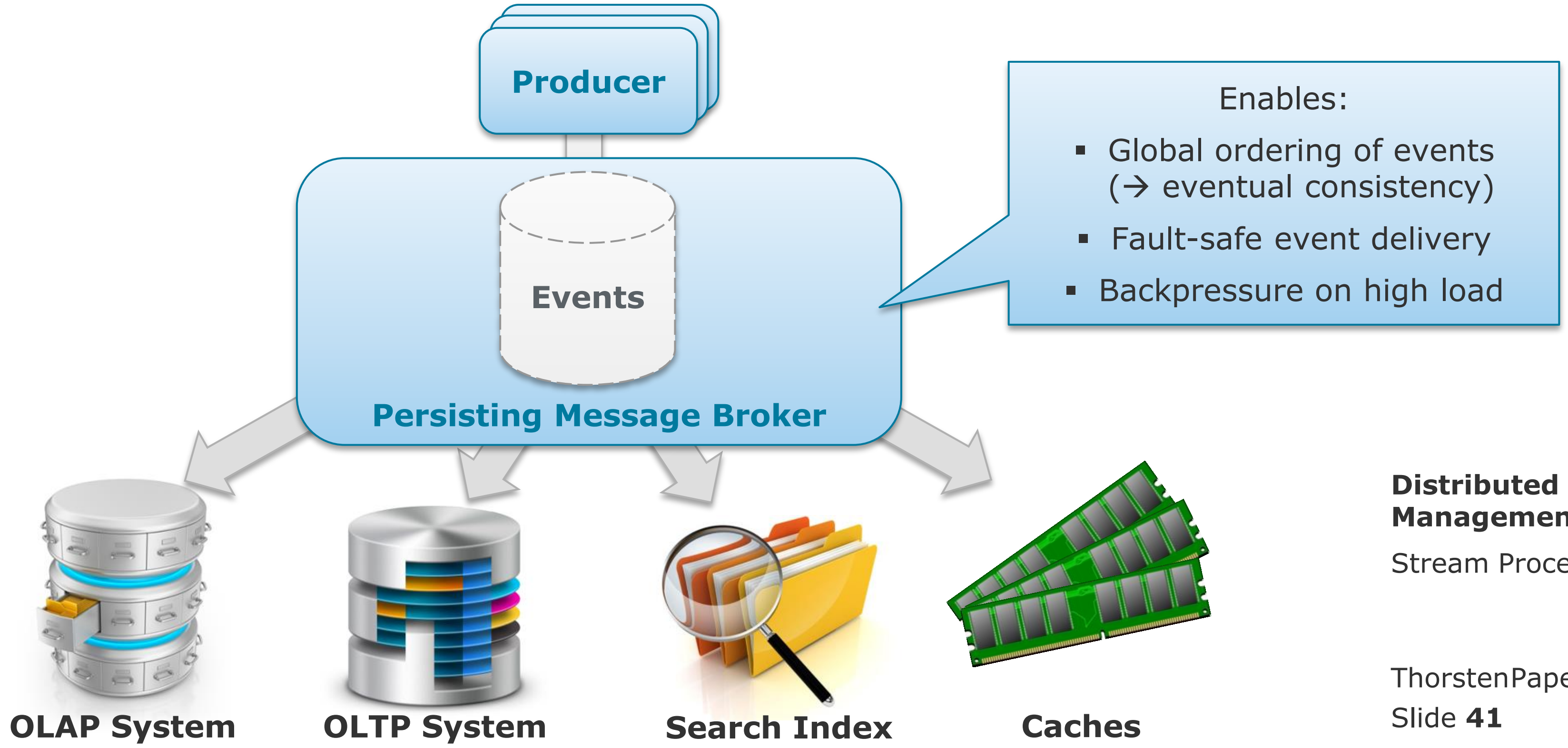
Data Storage – Keeping Systems in Sync



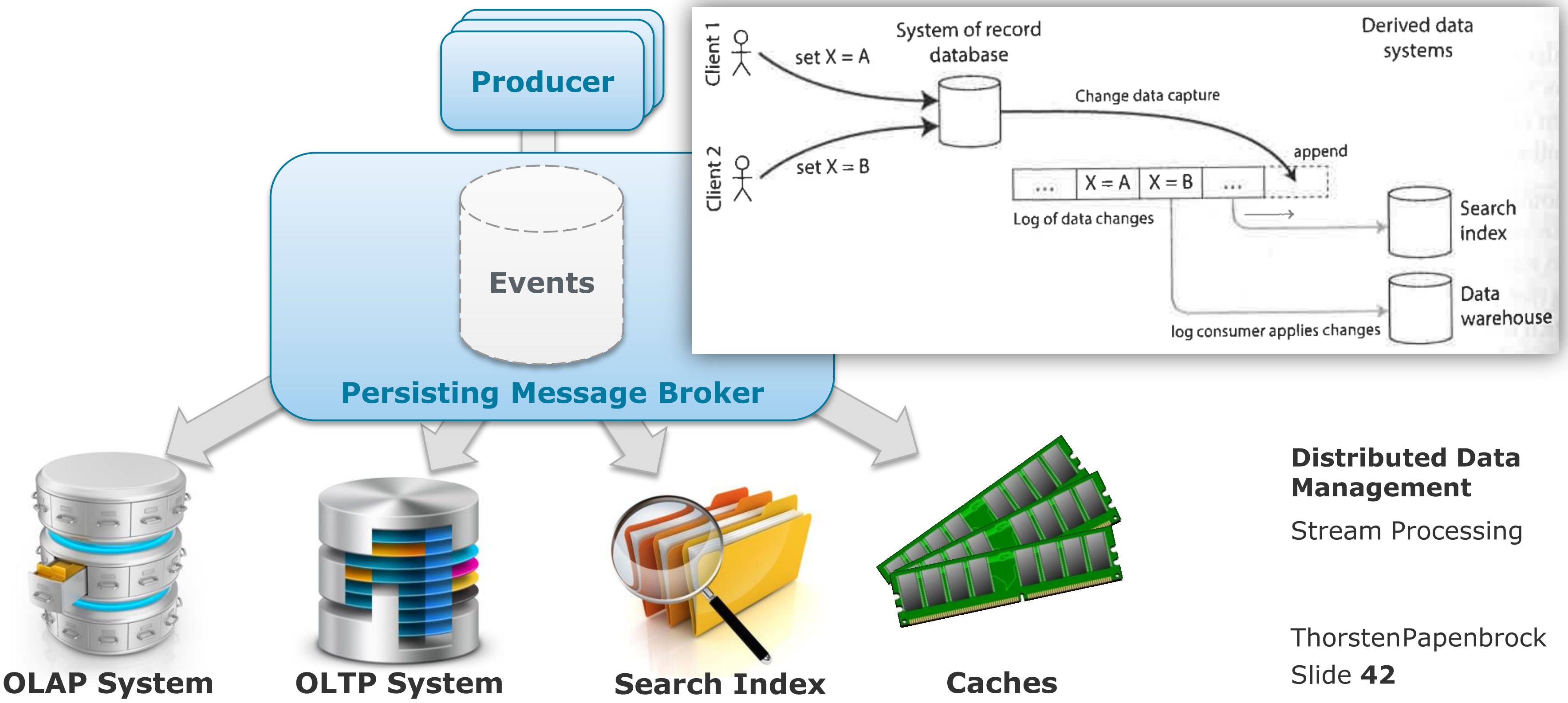
Write conflict:

Database and search index are inconsistent, because they don't share a common leader (that implements e.g. 2PC or MVCC).

Data Storage – Keeping Systems in Sync



Data Storage – Keeping Systems in Sync



Message Broker to Database

Data Change Event Streams

- If events are change operations (writes/deletes) to individual objects (records) it suffices to store only the most recent log entry for each object to rebuild a database.
- Log Compaction:
  - Periodically removes outdated log entries from the log
  - Lets the log grow linearly with the data

Message Broker → Database

- If the broker knows what the events mean (e.g. key-value mappings) it can apply log compaction.
  - ➢ Event log does not outgrow the maximum buffer size.
    - ➢ Message broker becomes a database.
- Implemented by e.g. Apache Kafka

Message Broker to Database

Message Broker as a Database

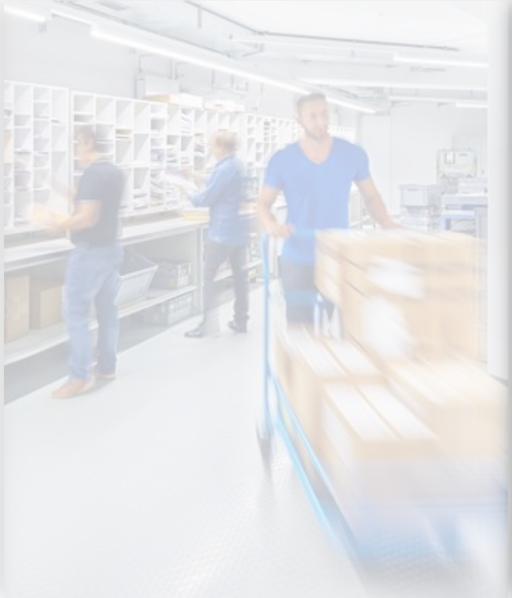
- Advantages:
  - Data Provenance/Auditability:
    - The line of events describes the history of every value.
    - ➢ Allows to follow a value back in time (e.g. the balance history of a bank account)
      - ➢ Fraud protection, temporal analytics, data recovery, …
  - Command Query Responsibility Segregation (CQRS):
    - Events describe what happened (= facts) not their implications.
    - ➢ Allows consumers to read/interpret events differently (= different views)
      - ➢ Multi-tenant systems, system evolution, data analytics, …
- Disadvantages:
  - Non-standing reads are slow (need to scan and interpret the entire event history).
  - Deleting data means declaring it deleted (actually deleting data is hard).

Overview

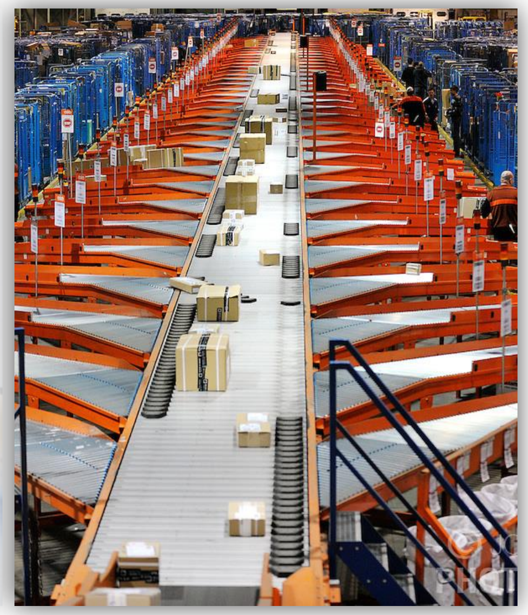
Stream Processing

Transmitting Event Streams

Databases and Streams



Processing Streams



Scenarios

Hasso Plattner Institut

Complex Event Processing (CEP)

- "Check a stream for patterns; whenever something special happens, raise a flag."
- Similar to pattern matching with regular expressions (often SQL-dialects)
- Implementations: Esper, IBM InfoSphere, Apama, TIBICO StreamBase, SQLstream

Stream Analytics

- "Transform or aggregate a stream; continuously output current results."
- Often uses statistical metrics and probabilistic algorithms:
  - Bloom filters (set membership)
  - HyperLogLog (cardinality estimation)
  - HDHistogram, t-digest, decay (percentile approximation)
- Implementations: Storm, Flink, Spark Streaming, Concord, Samza, Kafka Streams, Google Cloud Dataflow, Azure Stream Analytics

Bounded memory consumption

Approximation is often used for optimization, but Stream Processing is **not** inherently approximate!

Scenarios

Stream = Database
(using log compaction etc.)

Usually consider entire stream, i.e., no window!

Maintaining Materialized Views

- "Serve materialized views with up-to-date data from a stream."
- Views are also caches, search indexes, data warehouses, and any derived data system
- Implementations: Samza, Kafka Streams (but also works with Flink, Spark, and co.)

Search on Streams

- "Search for events in the stream; emit any event that matches the query."
- Similar to CEP but the standing queries are indexed, less complex, and more in number
- Implementations: Elasticsearch

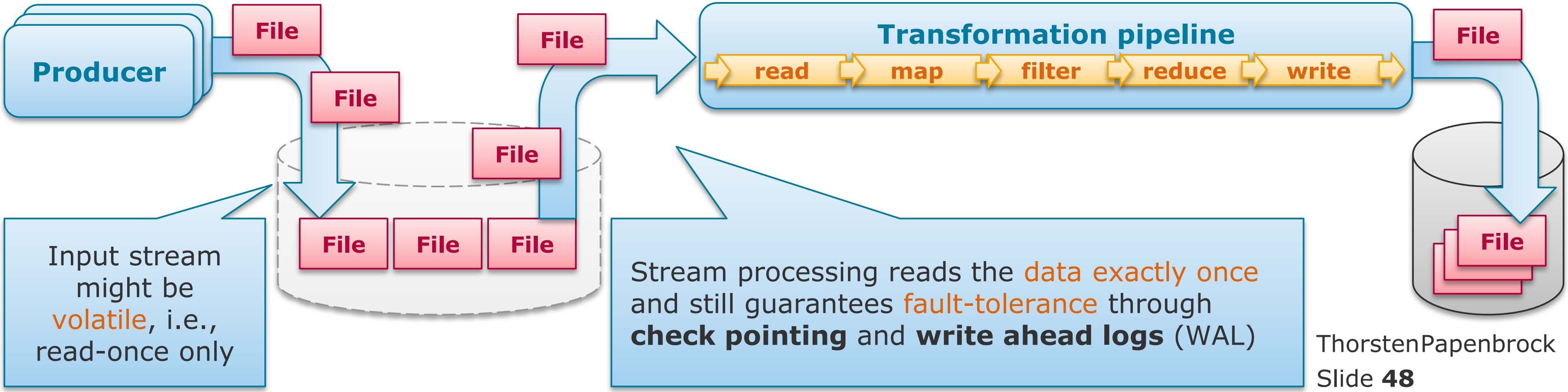
Message Passing

- "Use the stream for event communication; actors/processes consume and produce events."
- Requires non-blocking one-to-many communication
- Implementations: Any message broker; RPC systems with one-to-many support

Spark Streaming (Recap)

Batched Stream Processing

- Reasons:
  - Incremental processing: start processing data that is still being written to
  - Latency reduction: pipeline data to maximizing resource utilization



Examples

Spark Streaming (Recap)

```
val articles = spark
  .read
  .text("/mnt/data/articles/*.csv")


val words = articles.as[String].flatMap(_.split(" "))

val urls = words.filter(_.startsWith("http"))

val occurrences = urls.groupBy("value").count()


occurrences.show()
```

```
val articles = spark
  .readStream
  .text("/mnt/data/articles/*.csv")


val words = articles.as[String].flatMap(_.split(" "))

val urls = words.filter(_.startsWith("http"))

val occurrences = urls.groupBy("value").count()


val query = occurrences.writeStream
  .outputMode("complete")
  .format("console")
  .start()

query.awaitTermination()
```

Streaming input sources:

Files	text, csv, json, parquet
Kafka	Apache Kafka message broker
Socket	UTF8 text data from a socket
Rate	Generated data for testing

"complete"	write the entire result for every result update
"append"	append new results; old results should not change
"update"	output only changed results

Streaming output sinks:

Files	"parquet", "orc", "json", "csv", etc.
Kafka	"kafka" pointing to a Kafka topic
Foreach	.foreach(...)
Console	"console"
Memory	"memory" with .queryName("…")

STORM

Storm

- A free and open source distributed real-time computation system (stream processor)
- Competes with Apache Flink in stream processing speed
- Creates a directed acyclic graph (DAG) of "spout" and "bolt" vertices
  - Spout = streaming data source
  - Bolt = data transformation operator

- Designed for:
  - real-time analytics
  - online machine learning
  - continuous computation
  - distributed RPC
  - ETL

- Guarantees:
  - scalability
  - fault-tolerance
  - "best effort", "at least once", and "exactly once" processing capabilities
  - ease to set up and operate

Processing Streams

Examples

STORM


```
public class RandomSentenceSpout extends BaseRichSpout {
  SpoutOutputCollector _collector;
  Random _rand;


  @Override
  public void open(Map conf, TopologyContext context, SpoutOutputCollector collector) {
    _collector = collector;
    _rand = new Random();
  }

  @Override
  public void nextTuple() {
    Utils.sleep(100);
    String[] sentences = new String[]{ "the cow jumped over the moon", "an apple a day keeps the doctor away",
        "four score and seven years ago", "snow white and the seven dwarfs", "i am at two with nature" };
    String sentence = sentences[_rand.nextInt(sentences.length)];
    _collector.emit(new Values(sentence));
  }

  @Override
  public void ack(Object id) {
  }

  @Override
  public void fail(Object id) {
  }

  @Override
  public void declareOutputFields(OutputFieldsDeclarer declarer) {
    declarer.declare(new Fields("word"));
  }

}
```

A source that streams some text lines

Text to be streamed

Output format

http://admicloud.github.io/www/storm.html

Examples

STORM


```
public class RandomSentenceSpout extends BaseRichSpout {
  SpoutOutputCollector _collector;
  Random _rand;
```

```
public static class SplitSentence extends BaseBasicBolt {
    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
      declarer.declare(new Fields("word"));
    }

    @Override
    public Map<String, Object> getComponentConfiguration() {
      return null;
    }

    public void execute(Tuple tuple, BasicOutputCollector basicOutputCollector) {
      String sentence = tuple.getStringByField("sentence");
      String words[] = sentence.split(" ");
      for (String w : words) {
        basicOutputCollector.emit(new Values(w));
      }
    }
}
```

Storm bolds implement UDFs

A flatMap() implementation

```
  @Override
  public void fail(Object id) {
  }

  @Override
  public void declareOutputFields(OutputFieldsDeclarer declarer) {
    declarer.declare(new Fields("word"));
  }

}
```

http://admicloud.github.io/www/storm.html

Examples

STORM

Hasso Plattner Institut

http://admicloud.github.io/www/storm.html

```java
public class RandomSentenceSpout extends BaseRichSpout {
  SpoutOutputCollector _collector;
  Random _rand;
```

```java
public static class SplitSentence extends BaseBasicBolt {
    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
      declarer.declare(new Fields("word"));
```

```java
public static class WordCount extends BaseBasicBolt {
    Map<String, Integer> counts = new HashMap<String, Integer>();

    @Override
    public void execute(Tuple tuple, BasicOutputCollector collector) {
      String word = tuple.getString(0);
      Integer count = counts.get(word);
      if (count == null)
        count = 0;
      count++;
      counts.put(word, count);
      collector.emit(new Values(word, count));
    }

    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
      declarer.declare(new Fields("word", "count"));
    }
}
```

Another flatMap() implementation

Streaming output: emit every update

```java
  public void declareOutputFields(OutputFieldsDeclarer declarer) {
    declarer.declare(new Fields("word"));
  }

}
```

Examples

STORM

HPI Hasso Plattner Institut

http://admicloud.github.io/www/storm.html

More on Apache Storm @ http://storm.apache.org/

```
public class RandomSentenceSpout extends BaseRichSpout {
  SpoutOutputCollector _collector;
  Random _rand;
```

```
public static class SplitSentence extends BaseBasicBolt {
    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
      declarer.declare(new Fields("word"));
```

```
public static class WordCount extends BaseBasicBolt {
    Map<String, Integer> counts = new HashMap<String, Integer>();
```

```
public static void main(String[] args) throws Exception {
    TopologyBuilder builder = new TopologyBuilder();
    builder.setSpout("spout", new RandomSentenceSpout(), 5);
    builder.setBolt("split", new SplitSentence(), 8).shuffleGrouping("spout");
    builder.setBolt("count", new WordCount(), 12).fieldsGrouping("split", new Fields("word"));

    Config conf = new Config();
    conf.setDebug(true);

    if (args != null && args.length > 0) {
      conf.setNumWorkers(3);

      StormSubmitter.submitTopologyWithProgressBar(args[0], conf, builder.createTopology());
    } else {
      conf.setMaxTaskParallelism(3);
      LocalCluster cluster = new LocalCluster();
      cluster.submitTopology("word-count", conf, builder.createTopology());
      Thread.sleep(10000);
      cluster.shutdown();
    }
  }
}
```

Parallelism hint for spouts/bolts

Define the grouping for the input of each bolt:
- shuffle: assign randomly
- field: assign by field value

Execute on cluster

Execute locally

Runs until explicitly stopped

Processing Streams

Examples

STORM

Hasso Plattner Institut

http://admicloud.github.io/www/storm.html

```
public class RandomSentenceSpout extends BaseRichSpout {
  SpoutOutputCollector _collector;
  Random _rand;
```

```
public static class SplitSentence extends BaseBasicBolt {
    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
      declarer.declare(new Fields("word"));
```

```
public static class WordCount extends BaseBasicBolt {
    Map<String, Integer> counts = new HashMap<String, Integer>();

    @Override
    public void execute(Tuple tuple, BasicOutputCollector collector) {
      String word = tuple.getString(0);
      Integer count = counts.get(word);
      if (count == null)
        count = 0;
      count++;
      counts.put(word, count);
      collector.emit(new Values(word, count));
    }

    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
      declarer.declare(new Fields("word", "count"));
    }
}
```

In-memory data structure that grows indefinitely large

Implemented as a narrow flatMap() and not as a wide groupBy() to avoid blocking of the pipeline

```
      cluster.submitTopology("word-count", conf, builder.createTopology());
      Thread.sleep(10000);
      cluster.shutdown();
    }
}
```

Challenges and Limits

Goal

- Query and analyze streaming data in real-time (i.e. as data passes by).

Challenges

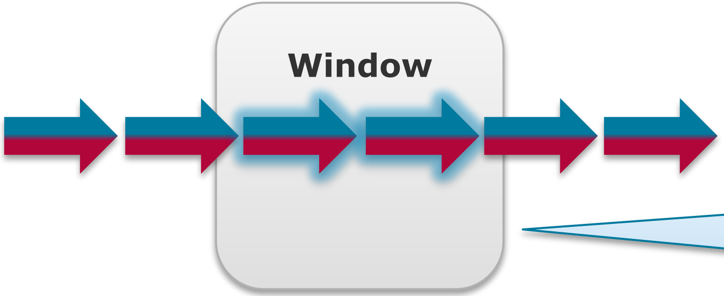
- Limited memory resources (but endlessly large volumes of data)
  - Only a fixed-size window of the stream is accessible at a time.
- Old data is permanently gone (and not accessible any more)
  - Only one-pass algorithms can be used.
- Endlessness contradicts certain operations
  - E.g. sorting makes no sense, i.e., no sort-merge-joins or groupings (on the entire stream!).
- Input cannot be re-read or easily back-traced
  - Fault tolerance must be ensured differently.

Concepts

Windows

- A continuous segment of the stream usually implemented as a buffer
  - New events oust the oldest events from the window.
- Events within the window can be accessed arbitrarily often.
- Bounded in size usually using a time interval or a maximum number of events

While sliding over the events, successive windows may or may not overlap

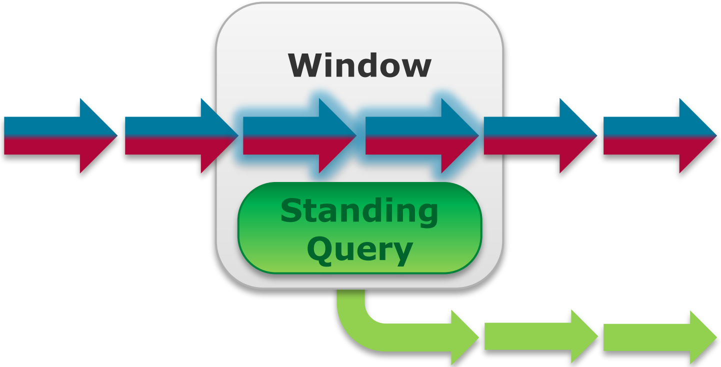


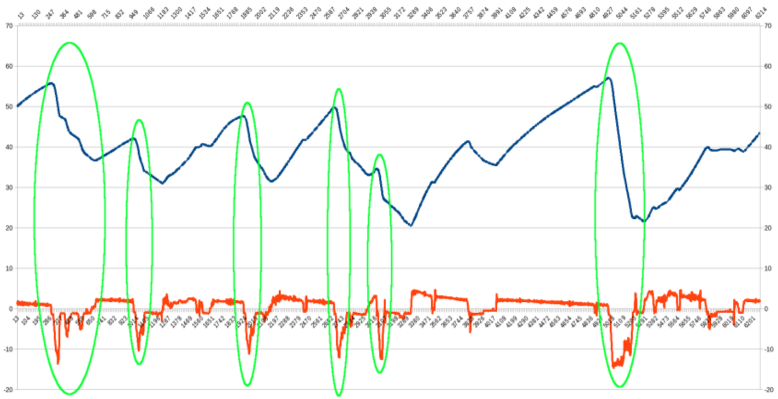
At the heart of processing infinite streams, as they let us make exact statements for concrete sub-sequences

Concepts

Standing queries

- Persisted queries that are served with volatile event data (reversed DBMS principle)
- Produce a streaming output of "complex events"
- Apply event checking, pattern matching, correlation analysis, aggregation, …
- Operate on windows





Windows

File-based micro-batching!

Tumbling Windows

- Fixed-length, non-overlapping windows
  → New window starts when previous window ended (e.g. successive intervals of 3 seconds or 100 events)

Hopping Windows

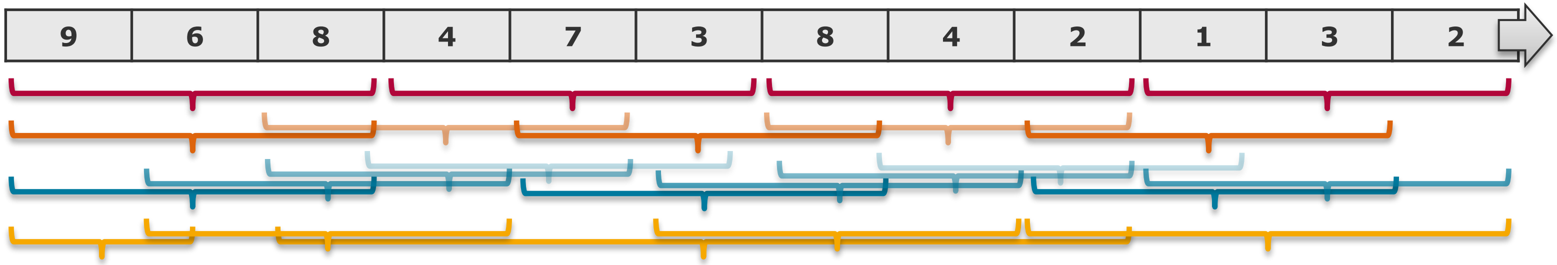
- Fixed-length, overlapping windows with fix steps
  → Defined by window length and hop width (e.g. intervals of 3 seconds starting every 2 seconds)

Sliding Windows

- Fixed-length, overlapping windows with event dependent steps
  → Either new events oust old events or events stay for a certain amount of time

Session Windows

- Arbitrary-length, overlapping windows
  → Fix start- and end-event (e.g. user logs in; user logs out or session times out)



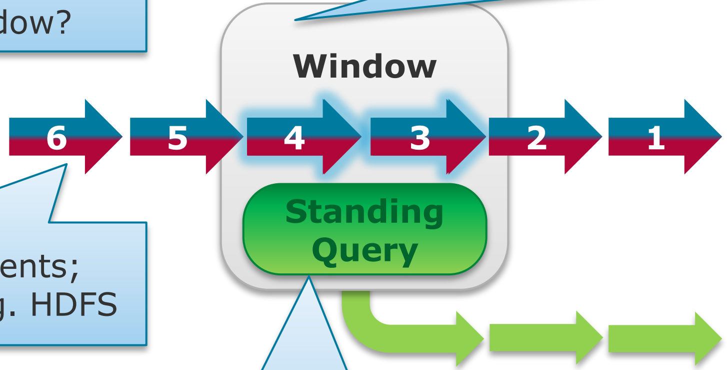
Processing Streams

Windows and Parallelization

How does parallelization happen?

Different windows can be processed in parallel, but how do we parallelize one window?

We expect a repartition() here, but for streaming scenarios and overlapping windows, this should be a stable operation in accordance with event/ingestion/processing time and order.



One input stream of events; not pre-partitioned by e.g. HDFS

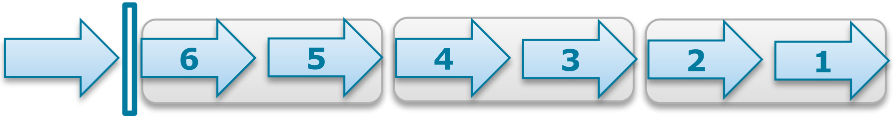
Process sequences of logically related events

The framework does not automatically know which elements belong together and which can be processed in parallel.

Windows and Parallelization

Non-Keyed Windows

- Partition a stream into another stream of buckets
- For parallel processing, events need to be replicated (not supported by all streaming frameworks)
  - ➢ Usually no parallelization without keying



Keyed Windows

- Partition a stream into multiple other streams of buckets (one per key value)
- Output streams can naturally be processed in parallel without replication
  - ➢ Default stream parallelization technique

Also called partitioned windows

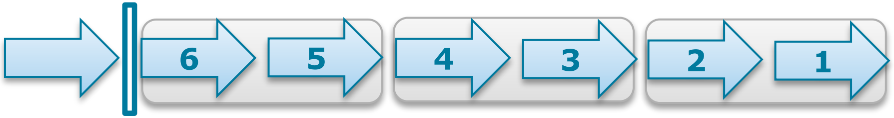
Processing Streams

Windows and Parallelization

Hasso Plattner Institut

Non-Keyed Windows

```
stream
   .windowAll(...)              <-  required: "assigner"
  [.trigger(...)]               <-  optional: "trigger" (else default trigger)
  [.evictor(...)]               <-  optional: "evictor" (else no evictor)
  [.allowedLateness(...)]       <-  optional: "lateness" (else zero)
  [.sideOutputLateData(...)] <-  optional: "output tag" (else no side output for late data)
   .reduce/aggregate/fold/apply()      <-  required: "function"
  [.getSideOutput(...)]         <-  optional: "output tag"
```



Keyed Windows

```
stream
   .keyBy(...)                  <-  keyed versus non-keyed windows
   .window(...)                 <-  required: "assigner"
  [.trigger(...)]               <-  optional: "trigger" (else default trigger)
  [.evictor(...)]               <-  optional: "evictor" (else no evictor)
  [.allowedLateness(...)]       <-  optional: "lateness" (else zero)
  [.sideOutputLateData(...)] <-  optional: "output tag" (else no side...)
   .reduce/aggregate/fold/apply()      <-  required: "function"
  [.getSideOutput(...)]         <-  optional: "output tag"
```

https://ci.apache.org/projects/flink/flink-docs-stable/dev/stream/operators/windows.html#triggers

Flink

Hasso Plattner Institut

Flink

```
val env = StreamExecutionEnvironment.getExecutionEnvironment
val text = env.socketTextStream("localhost", 4242, '\n')

val windowCounts = text
  .flatMap { w => w.split("\\s") }
  .map { w => WordWithCount(w, 1) }
  .keyBy("word")
  .timeWindow(Time.seconds(5), Time.seconds(1))
  .sum("count")
windowCounts.print().setParallelism(1)
env.execute("Socket Window WordCount")

case class WordWithCount(word: String, count: Long)
```

Get the execution environment

Get input data by connecting to the socket

Parse the data, map the words, and group them

Define a sliding window of size 5 seconds that slides every 1 second

Aggregate the counts per window

Print the results with a single thread, rather than in parallel

More on Apache Flink @ https://flink.apache.org/

Examples

CQL

Continuous Query Language

- Developed at Stanford University: http://www-db.stanford.edu/stream
- Used to define standing queries for windows of a stream

```
SELECT count(*)
FROM Requests R [RANGE 1 Day PRECEDING]
WHERE R.domain = 'stanford.edu'
```

stream

window (defined using time)

"Count the number of requests to stanford.edu for the last 1 day."

```
SELECT count(*)
FROM Requests R [PARTITION BY R.client_id
                 ROWS 10 PRECEDING
                 WHERE R.domain = 'stanford.edu']
WHERE R.url LIKE 'http://cs.stanford.edu/%'
```

partitioning (by attribute value)

window (defined using size)

"From the last 10 requests of a user to standord.edu, count all her calls to cs."

Events and Time

Event Time

- Creation time of the event on the producer (when it occurred)

Ingestion Time

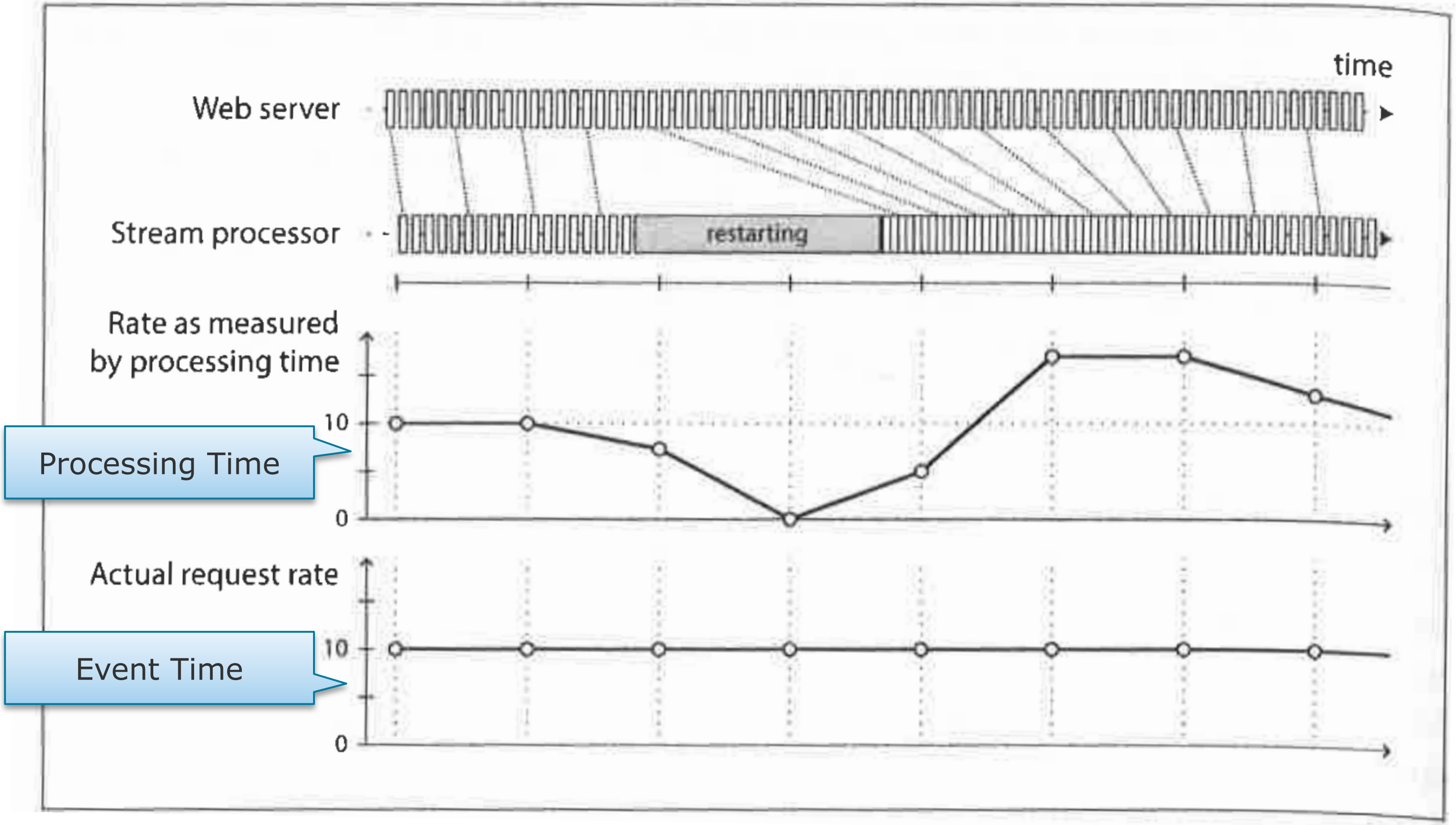
- Arrival time of the event at the stream processor (when it was received)

Processing Time

- Operation time of the event on the stream processor (when it had an effect)

Stream processors (e.g. Flink) let you choose which time to use for windowing!

Event Time vs. Processing Time



Events and Time

Event Time

- Creation time of the event on the producer (when it occurred)

Ingestion Time

- Arrival time of the event at the stream processor (when it was received)

Processing Time

- Operation time of the event on the stream processor (when it had an effect)

Unpredictable Time Lag

- Events might be delayed due to …
  - congestion, queuing, faults, …
- Events might be out-of-order due to …
  - message loss and resend, alternative routing, …
- Event time might be measured differently due to …
  - multiple clocks in distributed systems, clock skew and correction, …

Recall lecture on "Distributed Systems"

Processing Streams

Event Time vs. Processing Time

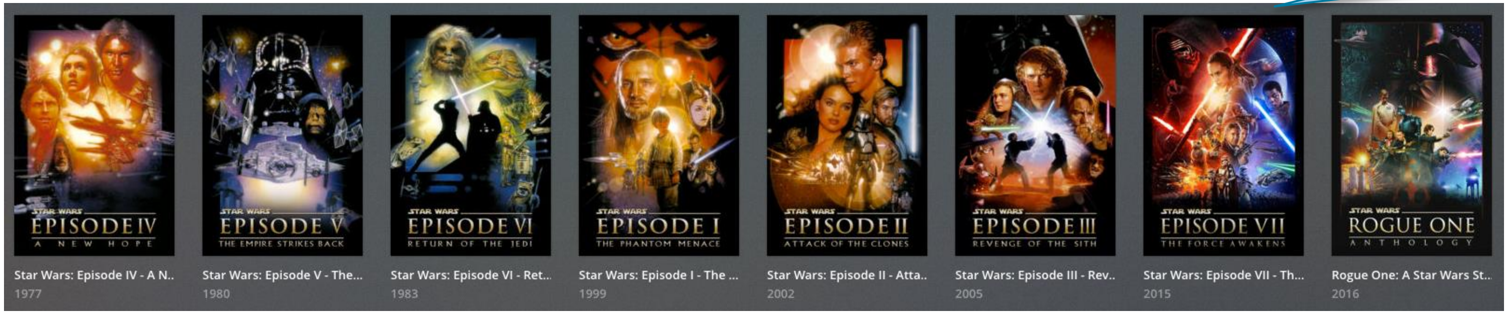
Solutions

- Assign timestamps as early as possible:
  - producer > leader > time-synced worker > un-synced worker
- Assign multiple timestamps
  - creation-time, send-time, receive-time, forward-time, …
- Solve time lag programmatically:
  - Exchange a fixed event frequency (e.g. frequency = 1 second)
  - Reasoning over events (e.g. order(X) > pay(X) > deliver(X))

Many events (e.g. sensor or log) carry timestamps naturally

Used to calculate the lag

filming order ≠ narrative order



Completing a Window

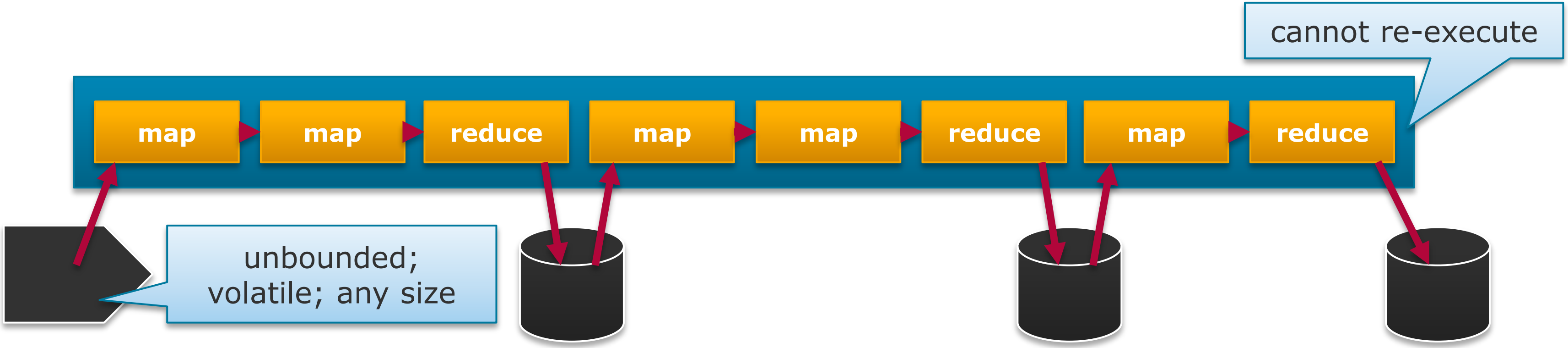
Problem

- How does a stream worker know that all events for a certain window have arrived?
  (as events might be delayed → straggler events)

Solution

- Declare a window as completed if …
  a) the first event for next window arrives or
  b) a timeout for this window has elapsed.
- Handle straggler events after completion of their window by …
  a) ignoring them (maybe counting/reporting ignored stragglers) or
  b) publishing an update for their window or
  c) assigning them to the next window.

Fault Tolerance



Issues

- Unbounded:
  - Jobs cannot wait making their output visible until their stream finishes
- Volatile:
  - If a fault occurs, stream data cannot be re-read

Fault Tolerance

Microbatching and Checkpointing

- Microbatches (see Spark):
  - Tumbling windows that are treated as batches (cached, checkpointed, …).
  - Windows represent state that is written to disk and serves to recover from faults.
- Checkpoints (see Flink):
  - Rolling checkpoints that are triggered periodically by barriers in the event stream.
  - Operator state is written to disk and serves to recover from faults.
  - Checkpoints are not tied to particular window sizes.
- Both strategies ensure that every event is processed
  - No event is lost until it produced some output.
- Still problematic:
  - Actions that recover from faults might produced redundant outputs to external event sinks (databases, message brokers, HDFS, …).

Fault Tolerance

Atomic Commit (revisited)

- Avoid **redundant outputs** using a commit protocol in conjunction with every event sink.
- Commits are logged, which helps to check whether an output happened before.
- Single event commits are cheaper than transaction commits.
- Still a research area with only a few systems supporting it:
    - Google Cloud Dataflow, VoltDB, Kafka (in development)

Idempotence

- Avoid **redundant output effects** using only idempotent output operations.
- Idempotent operation = operation that has the same effect regardless how often it is applied.
- Examples (multiple calls always replace the existing data with itself):
    - Set key to value; Create file with name; Delete resource; Overwrite content with text
- Many non-idempotent operations can be made idempotent:
    - Add an offset/identifier to each output event that identifies redundancy.

Joins

Stream-Stream Join

- Task: Join events in stream A with events in stream B.
- Problem: Joins require all events of one side to be randomly accessible, but stream is endless.
- Solution: Window Joins
  - One side of the join is kept in memory as a window
    (e.g. session window of logged-in users).
  - The other side of the join is probed against the events of that window
    (e.g. request events to an API).
  - Straggler events are dropped.

Stream-Table Join

- Task: Join events in a stream with events in a database.
- Problem: Database is too large for memory and too slow for stream checks.
- Solution: Database Partitioning/Replication
  - Forward the stream to different partitions/replica that perform different parts of the join.

Further Reading

T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle. *The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing*. Proceedings of the VLDB Endowment 8, 12 (August 2015), 1792-1803. DOI=http://dx.doi.org/10.14778/2824032.2824076

The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing

Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, Sam Whittle
Google
{takidau, robertwb, chambers, chernyak, rfernand, relax, sgmc, millsd, fjp, cloude, samuelw}@google.com

ABSTRACT

Unbounded, unordered, global-scale datasets are increasingly common in day-to-day business (e.g. Web logs, mobile usage statistics, and sensor networks). At the same time, consumers of these datasets have evolved sophisticated requirements, such as event-time ordering and windowing by features of the data themselves, in addition to an insatiable hunger for faster answers. Meanwhile, practicality dictates that one can never fully optimize along all dimensions of correctness, latency, and cost for these types of input. As a result, data processing practitioners are left with the quandary of how to reconcile the tensions between these seemingly competing propositions, often resulting in disparate implementations and systems.

We propose that a fundamental shift of approach is necessary to deal with these evolved requirements in modern data processing. We as a field must stop trying to groom unbounded datasets into finite pools of information that eventually become complete, and instead live and breathe under the assumption that we will never know if or when we have seen all of our data, only that new data will arrive, old data may be retracted, and the only way to make this problem tractable is via principled abstractions that allow the practitioner the choice of appropriate tradeoffs along the axes of interest: correctness, latency, and cost.

In this paper, we present one such approach, the Dataflow Model[1], along with a detailed examination of the semantics it enables, an overview of the core principles that guided its design, and a validation of the model itself via the real-world experiences that led to its development.

[1]We use the term "Dataflow Model" to describe the processing model of Google Cloud Dataflow [20], which is based upon technology from FlumeJava [12] and MillWheel [2].

Proceedings of the VLDB Endowment, Vol. 8, No. 12

1. INTRODUCTION

Modern data processing is a complex and exciting field. From the scale enabled by MapReduce [16] and its successors (e.g Hadoop [4], Pig [18], Hive [29], Spark [33]), to the vast body of work on streaming within the SQL community (e.g. query systems [1, 14, 15], windowing [22], data streams [24], time domains [28], semantic models [9]), to the more recent forays in low-latency processing such as Spark Streaming [34], MillWheel, and Storm [5], modern consumers of data wield remarkable amounts of power in shaping and taming massive-scale disorder into organized structures with far greater value. Yet, existing models and systems still fall short in a number of common use cases.

Consider an initial example: a streaming video provider wants to monetize their content by displaying video ads and billing advertisers for the amount of advertising watched. The platform supports online and offline views for content and ads. The video provider wants to know how much to bill each advertiser each day, as well as aggregate statistics about the videos and ads. In addition, they want to efficiently run offline experiments over large swaths of historical data.

Advertisers/content providers want to know how often and for how long their videos are being watched, with which content/ads, and by which demographic groups. They also want to know how much they are being charged/paid. They want all of this information as quickly as possible, so that they can adjust budgets and bids, change targeting, tweak campaigns, and plan future directions in as close to real time as possible. Since money is involved, correctness is paramount.

Though data processing systems are complex by nature, the video provider wants a programming model that is simple and flexible. And finally, since the Internet has so greatly expanded the reach of any business that can be parceled along its backbone, they also require a system that can handle the diaspora of global scale data.

The information that must be calculated for such a use case is essentially the time and length of each video viewing, who viewed it, and with which ad or content it was paired (i.e. per-user, per-video viewing *sessions*). Conceptually this is straightforward, yet existing models and systems all fall short of meeting the stated requirements.

Batch systems such as MapReduce (and its Hadoop variants, including Pig and Hive), FlumeJava, and Spark suffer

1792

https://ci.apache.org/projects/flink/flink-docs-release-1.6/

Flink v1.6

- Home
- Concepts
- Quickstart
- Examples
- Project Setup
- Application Development
- Deployment & Operations
- Debugging & Monitoring
- Internals
- Javadocs
- Scaladocs
- Project Page

Search

Go

Pick Docs Version

Home

Apache Flink Documentation

This documentation is for Apache Flink version 1.6. These pages were built at: 01/16/19, 02:01:50 AM UTC.

Apache Flink is an open source platform for distributed stream and batch data processing. Flink's core is a streaming dataflow engine that provides data distribution, communication, and fault tolerance for distributed computations over data streams. Flink builds batch processing on top of the streaming engine, overlaying native iteration support, managed memory, and program optimization.

First Steps

- **Concepts**: Start with the basic concepts of Flink's Dataflow Programming Model and Distributed Runtime Environment. This will help you understand other parts of the documentation, including the setup and programming guides. We recommend you read these sections first.
- **Quickstarts**: Run an example program on your local machine or study some examples.
- **Programming Guides**: You can read our guides about basic API concepts and the DataStream API or the DataSet API to learn how to write your first Flink programs.

Deployment

Before putting your Flink job into production, read the Production Readiness Checklist.

Release Notes

Release notes cover important changes between Flink versions. Please carefully read these notes if you plan to upgrade your Flink setup to a later version.

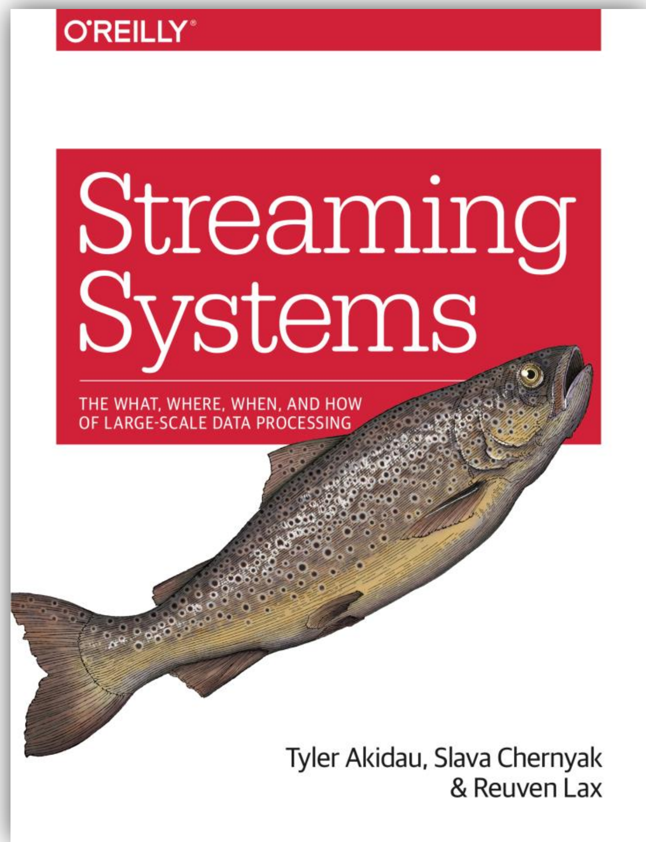
- Release notes for Flink 1.6.
- Release notes for Flink 1.5.

External Resources

- **Flink Forward**: Talks from past conferences are available at the Flink Forward website and on YouTube. Robust Stream Processing with Apache Flink is a good place to start.
- **Training**: The training materials from data Artisans include slides, exercises, and sample solutions.
- **Blogs**: The Apache Flink and data Artisans blogs publish frequent, in-depth technical articles about Flink.

Processing Streams
Further Reading

https://www.oreilly.com/ideas/the-world-beyond-batch-streaming-101



Streaming 101: The world beyond batch

A high-level tour of modern data-processing concepts.

By Tyler Akidau. August 5, 2015

The call for proposals is now open for the Strata Data Conference in London, April 29-May 2, 2019.

Editor's note: This is the first post in a two-part series about the evolution of data processing, with a focus on streaming systems, unbounded data sets, and the future of big data. See part two. Also, check out "Streaming Systems," by Tyler Akidau, Slava Chernyak, and Reuven Lax.

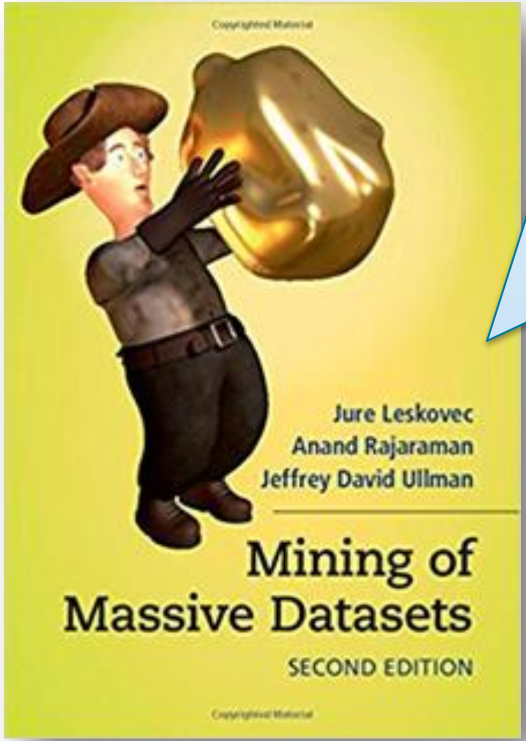
Three women wading in a stream gathering leeches (source: Wellcome Library, London)

Streaming data processing is a big deal in big data these days, and for good reasons. Amongst them:

- Businesses crave ever more timely data, and switching to streaming is a good way to achieve lower latency.
- The massive, unbounded data sets that are increasingly common in modern business are more easily tamed using a system designed for such never-ending volumes of data.
- Processing data as they arrive spreads workloads out more evenly over time, yielding more consistent and predictable consumption of resources.

Despite this business-driven surge of interest in streaming, the majority of streaming systems in existence remain relatively immature compared to their batch brethren, which has resulted in a lot of exciting, active development in the space recently.

Further Reading



1. Data Mining
2. Large-Scale File Systems and Map-Reduce
3. Finding Similar Items
4. Mining Data Streams
   - Sampling and Filtering
   - Counting and Aggregation
   - Estimation
   - Decaying Windows
5. Link Analysis
6. Frequent Itemsets
7. Clustering
8. Advertising on the Web
9. Recommendation Systems

Stream Processing
Check yourself

Given is a stream of elements $e_1, \ldots, e_n$. The task is to select a random sample of k elements (k <= n) from the stream, where each element of the stream should have the same probability to be sampled. The size of the stream is not known in advance.

Give an algorithm that solves this problem with O(k) memory and show that each element has the same probability to be sampled.

Homework

Log Data Analytics

```
thorsten@tody ~/Desktop $ head -n 50 access_log_Aug95
in24.inetnebr.com - - [01/Aug/1995:00:00:01 -0400] "GET /shuttle/missions/sts-68/news/sts-68-mcc-05.txt HTTP/1.0" 200 1839
uplherc.upl.com - - [01/Aug/1995:00:00:07 -0400] "GET / HTTP/1.0" 304 0
uplherc.upl.com - - [01/Aug/1995:00:00:08 -0400] "GET /images/ksclogo-medium.gif HTTP/1.0" 304 0
uplherc.upl.com - - [01/Aug/1995:00:00:08 -0400] "GET /images/MOSAIC-logosmall.gif HTTP/1.0" 304 0
uplherc.upl.com - - [01/Aug/1995:00:00:08 -0400] "GET /images/USA-logosmall.gif HTTP/1.0" 304 0
ix-esc-ca2-07.ix.netcom.com - - [01/Aug/1995:00:00:09 -0400] "GET /images/launch-logo.gif HTTP/1.0" 200 1713
uplherc.upl.com - - [01/Aug/1995:00:00:10 -0400] "GET /images/WORLD-logosmall.gif HTTP/1.0" 304 0
slppp6.intermind.net - - [01/Aug/1995:00:00:10 -0400] "GET /history/skylab/skylab.html HTTP/1.0" 200 1687
piweba4y.prodigy.com - - [01/Aug/1995:00:00:10 -0400] "GET /images/launchmedium.gif HTTP/1.0" 200 11853
slppp6.intermind.net - - [01/Aug/1995:00:00:11 -0400] "GET /history/skylab/skylab-small.gif HTTP/1.0" 200 9202
slppp6.intermind.net - - [01/Aug/1995:00:00:12 -0400] "GET /images/ksclogosmall.gif HTTP/1.0" 200 3635
ix-esc-ca2-07.ix.netcom.com - - [01/Aug/1995:00:00:12 -0400] "GET /history/apollo/images/apollo-logo1.gif HTTP/1.0" 200 1173
slppp6.intermind.net - - [01/Aug/1995:00:00:13 -0400] "GET /history/apollo/images/apollo-logo.gif HTTP/1.0" 200 3047
uplherc.upl.com - - [01/Aug/1995:00:00:14 -0400] "GET /images/NASA-logosmall.gif HTTP/1.0" 304 0
133.43.96.45 - - [01/Aug/1995:00:00:16 -0400] "GET /shuttle/missions/sts-69/mission-sts-69.html HTTP/1.0" 200 10566
kgtyk4.kj.yamagata-u.ac.jp - - [01/Aug/1995:00:00:17 -0400] "GET / HTTP/1.0" 200 7280
kgtyk4.kj.yamagata-u.ac.jp - - [01/Aug/1995:00:00:18 -0400] "GET /images/ksclogo-medium.gif HTTP/1.0" 200 5866
d0ucr6.fnal.gov - - [01/Aug/1995:00:00:19 -0400] "GET /history/apollo/apollo-16/apollo-16.html HTTP/1.0" 200 2743
ix-esc-ca2-07.ix.netcom.com - - [01/Aug/1995:00:00:19 -0400] "GET /shuttle/resources/orbiters/discovery.html HTTP/1.0" 200 6849
d0ucr6.fnal.gov - - [01/Aug/1995:00:00:20 -0400] "GET /history/apollo/apollo-16/apollo-16-patch-small.gif HTTP/1.0" 200 14897
kgtyk4.kj.yamagata-u.ac.jp - - [01/Aug/1995:00:00:21 -0400] "GET /images/NASA-logosmall.gif HTTP/1.0" 304 0
kgtyk4.kj.yamagata-u.ac.jp - - [01/Aug/1995:00:00:21 -0400] "GET /images/MOSAIC-logosmall.gif HTTP/1.0" 304 0
kgtyk4.kj.yamagata-u.ac.jp - - [01/Aug/1995:00:00:22 -0400] "GET /images/USA-logosmall.gif HTTP/1.0" 304 0
kgtyk4.kj.yamagata-u.ac.jp - - [01/Aug/1995:00:00:22 -0400] "GET /images/WORLD-logosmall.gif HTTP/1.0" 304 0
133.43.96.45 - - [01/Aug/1995:00:00:22 -0400] "GET /images/KSC-logosmall.gif HTTP/1.0" 200 1204
133.43.96.45 - - [01/Aug/1995:00:00:23 -0400] "GET /shuttle/missions/sts-69/sts-69-patch-small.gif HTTP/1.0" 200 8083
133.43.96.45 - - [01/Aug/1995:00:00:23 -0400] "GET /images/launch-logo.gif HTTP/1.0" 200 1713
www-c8.proxy.aol.com - - [01/Aug/1995:00:00:24 -0400] "GET /shuttle/countdown/ HTTP/1.0" 200 4324
133.43.96.45 - - [01/Aug/1995:00:00:25 -0400] "GET /history/apollo/images/apollo-logo1.gif HTTP/1.0" 200 1173
ix-esc-ca2-07.ix.netcom.com - - [01/Aug/1995:00:00:25 -0400] "GET /shuttle/resources/orbiters/discovery-logo.gif HTTP/1.0" 200 4179
piweba4y.prodigy.com - - [01/Aug/1995:00:00:32 -0400] "GET /images/NASA-logosmall.gif HTTP/1.0" 200 786
slppp6.intermind.net - - [01/Aug/1995:00:00:32 -0400] "GET /history/skylab/skylab-1.html HTTP/1.0" 200 1659
ix-esc-ca2-07.ix.netcom.com - - [01/Aug/1995:00:00:34 -0400] "GET /images/ksclogosmall.gif HTTP/1.0" 200 3635
in24.inetnebr.com - - [01/Aug/1995:00:00:34 -0400] "GET /shuttle/missions/sts-68/news/sts-68-mcc-06.txt HTTP/1.0" 200 2303
slppp6.intermind.net - - [01/Aug/1995:00:00:39 -0400] "GET /history/skylab/skylab-logo.gif HTTP/1.0" 200 3274
ix-esc-ca2-07.ix.netcom.com - - [01/Aug/1995:00:00:39 -0400] "GET /shuttle/resources/orbiters/orbiters-logo.gif HTTP/1.0" 200 1932
uplherc.upl.com - - [01/Aug/1995:00:00:43 -0400] "GET /shuttle/missions/sts-71/mission-sts-71.html HTTP/1.0" 200 13450
uplherc.upl.com - - [01/Aug/1995:00:00:44 -0400] "GET /shuttle/missions/sts-71/sts-71-patch-small.gif HTTP/1.0" 200 12054
uplherc.upl.com - - [01/Aug/1995:00:00:45 -0400] "GET /images/KSC-logosmall.gif HTTP/1.0" 200 1204
```

Log Data Analytics

Assignment

- Task
  - Data Exploration: Find interesting insights in a log stream, such as
    - the 90th percentile response size
    - average number of requests per hour
    - most popular clients and resources
  - Don’t break the memory!
- Dataset
  - Two month's worth of all HTTP requests to the NASA Kennedy Space Center WWW server in Florida: http://ita.ee.lbl.gov/html/contrib/NASA-HTTP.html
- Parameter
  - “java -jar YourAlgorithmName.jar --path access_log_Aug95 --cores 4”
  - Default path should be “./access_log_Aug95” and default cores 4

Inclusion Dependency Discovery - Rules

Assignment

- Expected output
  - Write your discoveries (text + value) to the console
  - Use the following style for your output:
    `<text> : <value>`
    - Example output:

```
90th percentile response size : 7265
average number of requests per hour : 233
most popular client : www.hpi.de
most popular resource : www.hpi.de/DDM
```

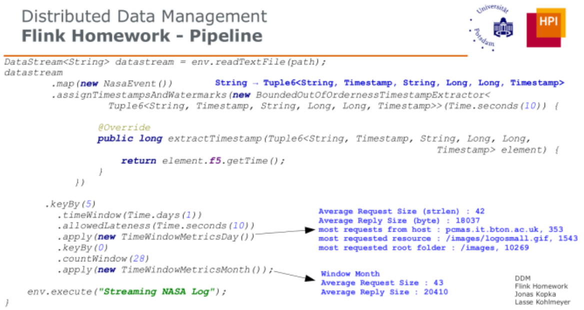
Inclusion Dependency Discovery - Rules

Assignment

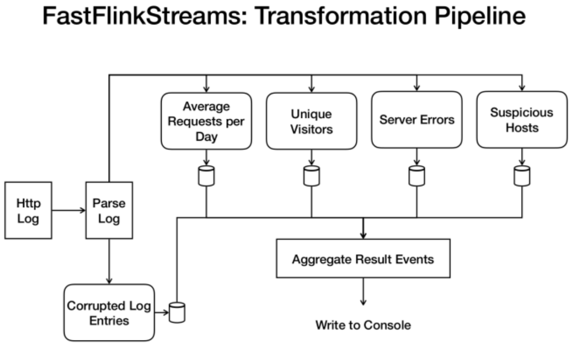
- Submission deadline
  - 27.01.2019 23:59:59
- Submission channel
  - ftp-share that we make available via email
- Submission artifacts
  - Source code as zip (Maven project; Java or Scala)
  - Jar file as zip (fat-jar)
  - a slide with your transformation pipeline(s)
- Teams
  - Please solve the homework in teams of two students
  - Provide the names of both students in your submission (= folder name)

- *text* = readTextFile
- .flatMap over lines in *text*:
  a. check if regex matches on line
  b. return matched groups as tuple
- .countWindowAll(100000)
  a. split into 100k chunks
- .process
  a. Turn current chunk into list
  b. Perform individual analysis
     i. Group by HTTP status, find count of 200 and non-200
     ii. Group by clients, find most common client
        1. Group paths for this client, find most common path
     iii. Group by path, sum sizes to find path with max traffic usage

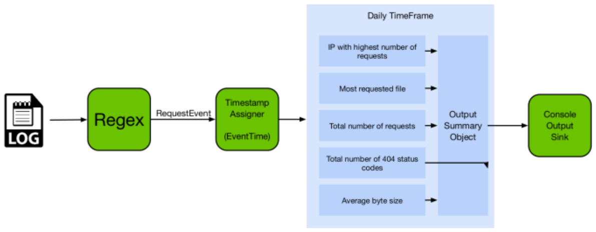
Team: **Most Metrics (Size Window)**



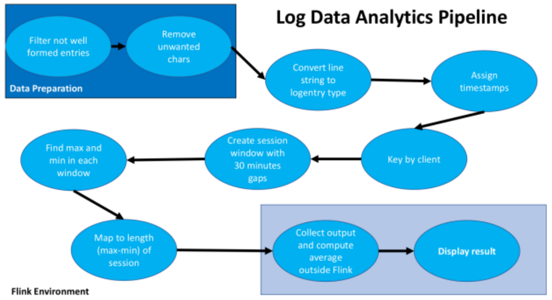
Team: **Most Metrics (Time Window)**



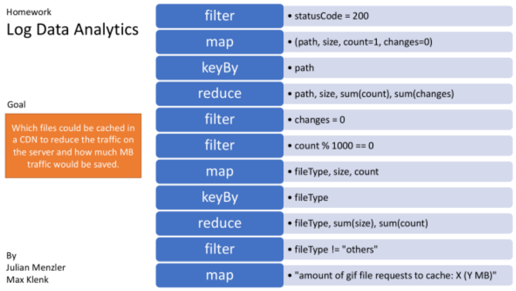
Team: **Disc Writing (Time Window)**



Team: **Output Summary (Time Window)**



Team: **Client Analytics (Keyed Session Window)**



Team: **Nice Use Case (Keyed Window)**


Sea of Derived Data
Materialized View Maintenance
ETL
Spark Streaming
Storm
Dataflow
Stream Analytics
The Great Event Stream
Flink
The Window
The Old Clock Tower
Sands of Time
Esper
CEP
To Databases
Kafka Streams
Samza
Mountains of State
Event sourcing
Kafka
Kinesis
AMQP
JMS
Message Queues
Forest of Logs
Change data capture
Sensors